Introduction to Deep Learning

Giovanni Della Lunga
giovanni.dellalunga@gmail.com

ADVANCED TOPICS IN ARTIFICIAL INTELLIGENCE

Bologna March 04-22, 2024

We will talk about...

In this lesson, we will cover the following topics:

- Introduction to Deep Learning. We will start by introducing the concept of deep learning, its history, and its applications.
- What is a Neural Network? We will delve into the fundamental building block of deep learning, the neural network. We will cover the architecture of a neural network and its different components.
- The Feed Forward Architecture. We will explore the simplest type of neural network, the feed-forward architecture, and learn how it can be used to solve a variety of problems.
- We will introduce Keras, a popular deep learning library, and learn how to use it to build and train neural networks.
- We will cover the basics of sequential data, such as time series, and explore how neural networks can be used to model and predict such data.

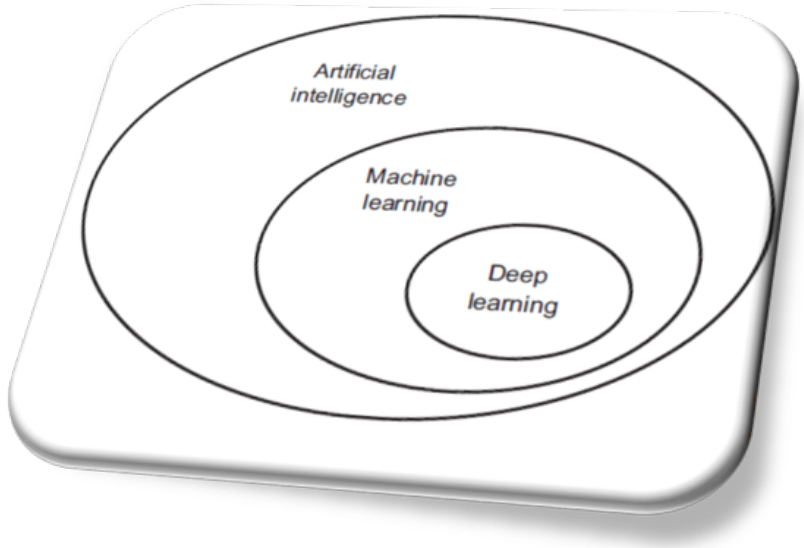
What is Deep Learning?

Subsection 1

Machine Learning and Deep Learning

Introduction: Machine Learning and Deep Learning

- As we know to do machine learning we need three things:
- Input data points
- Examples of the expected output
- A way to measure whether the algorithm is doing a good job



Introduction: Machine Learning and Deep Learning

- A machine-learning model transforms its input data into meaningful outputs, a process that is **learned** from exposure to known examples of inputs and outputs.
- Therefore, the central problem in machine learning and deep learning is to **meaningfully transform data**: in other words, **to learn useful representations of the input data at hand, representations that get us closer to the expected output**.
- What is a representation? At its core, it is simply a different way to look at data, to represent or encode data.

Machine Learning and Deep Learning

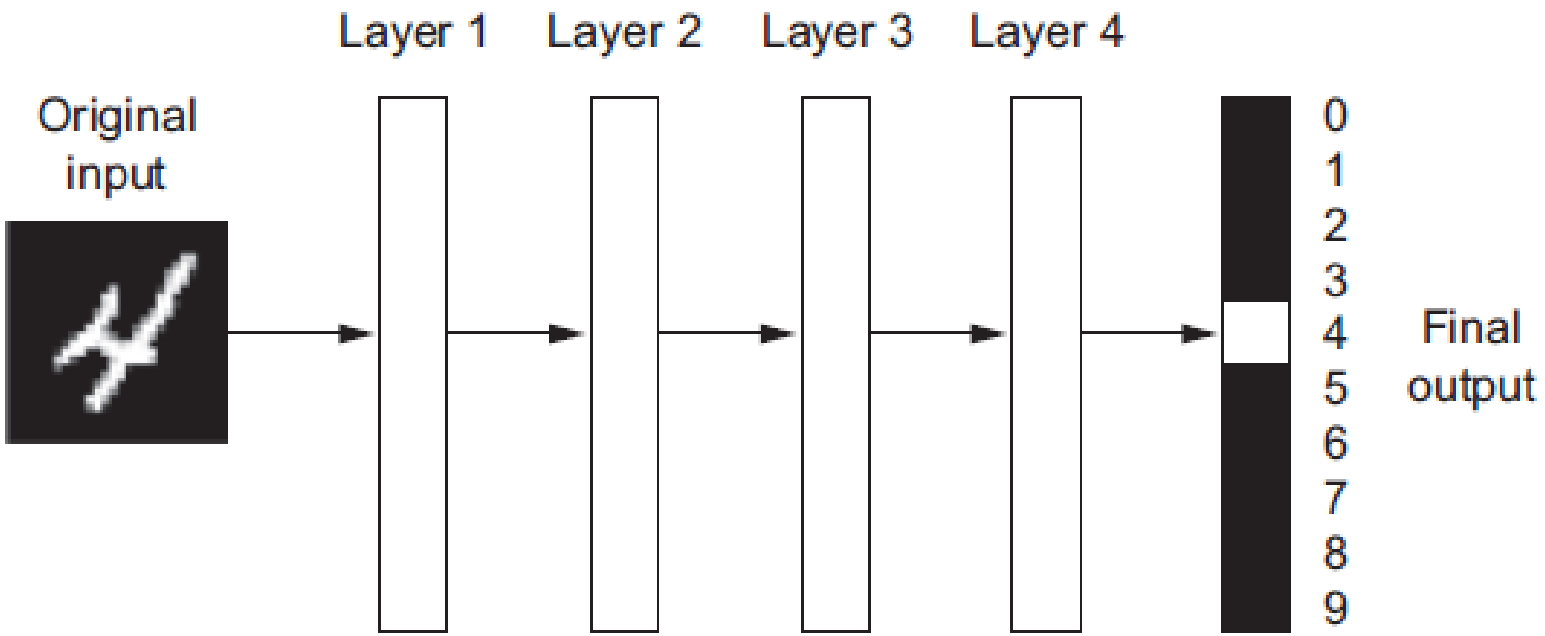
- From a more formal poin of view, deep learning learning process involves input variables, which we call X, and output variables, which we call Y.
- We use neural network to learn the mapping function from the input to the output.
- In simple mathematics, the output Y is a dependent variable of input X as illustrated by:

$$Y = f(X)$$

Here, our end goal is to try to **approximate the mapping function** f, so that we can **predict** the output variables Y when we have new input data X.

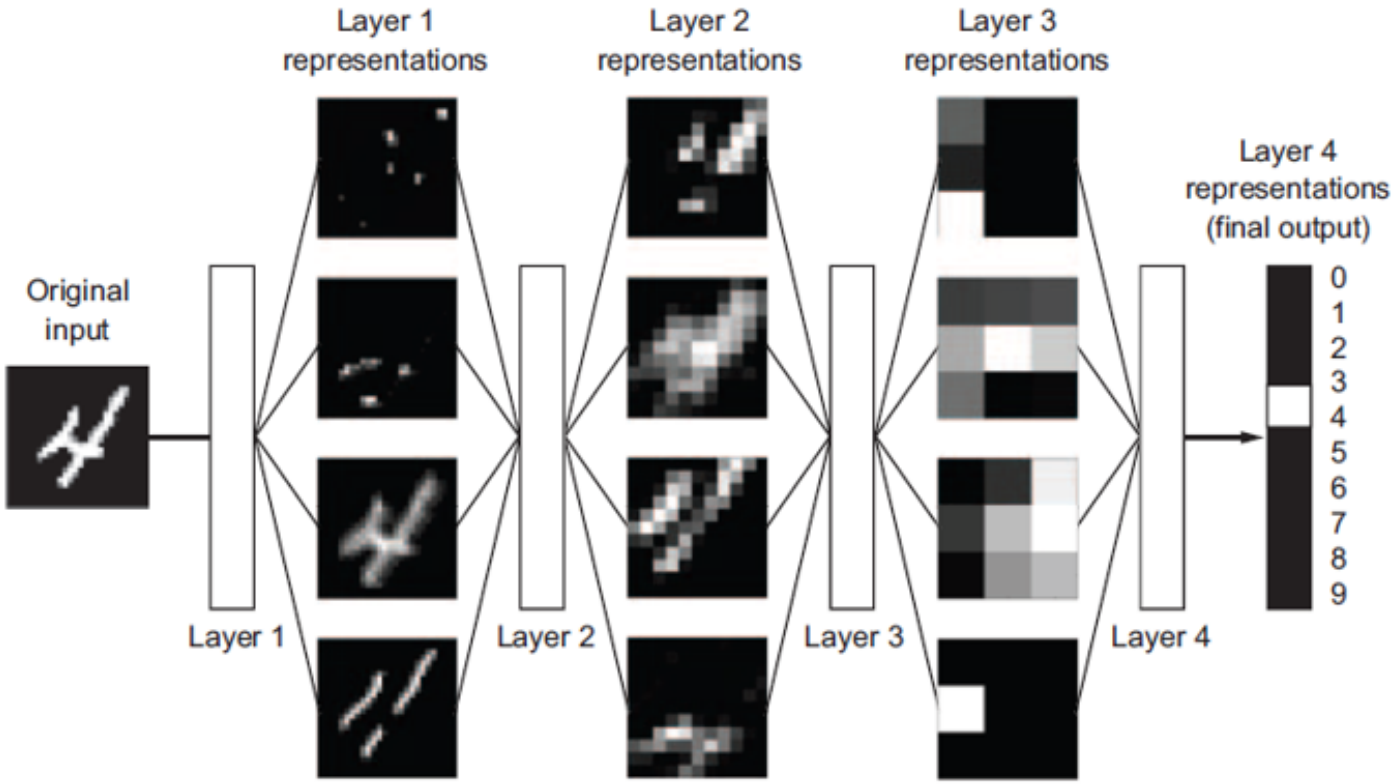
Introduction: Machine Learning and Deep Learning

Deep learning is a specific subfield of machine learning: a new take on learning representations from data that puts an emphasis on learning successive layers of increasingly meaningful representations.



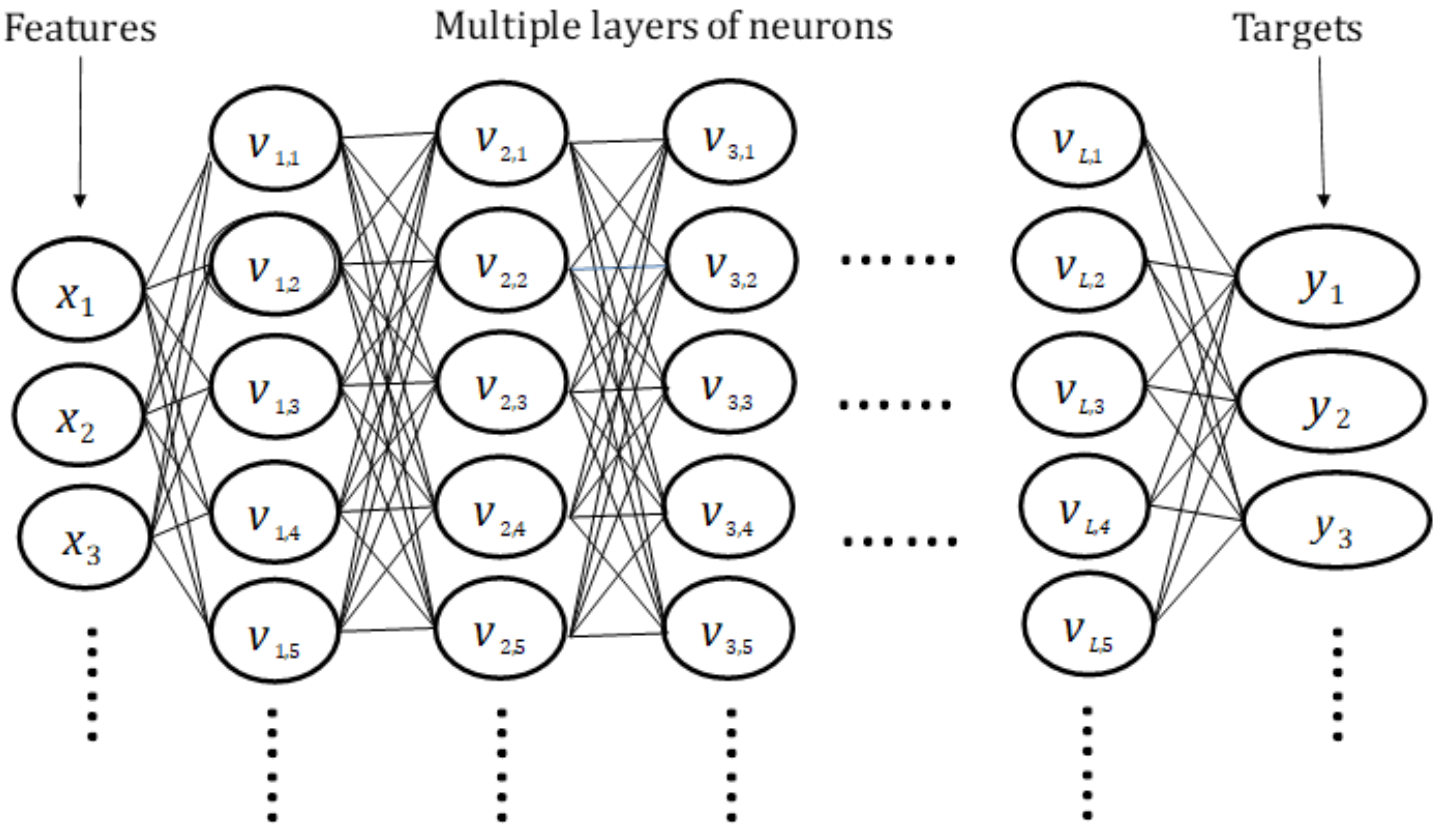
Introduction: Machine Learning and Deep Learning

Neural networks are typically composed of several layers of interconnected nodes, with each layer performing a specific transformation on the data. The input data is fed into the first layer, and the output of that layer becomes the input to the next layer, and so on, until the final output is produced...



Introduction: Machine Learning and Deep Learning

Neural networks are typically composed of several layers of interconnected nodes, with each layer performing a specific transformation on the data. The input data is fed into the first layer, and the output of that layer becomes the input to the next layer, and so on, until the final output is produced...



Neural Networks

Subsection 1

The McCulloch-Pitts Neuron

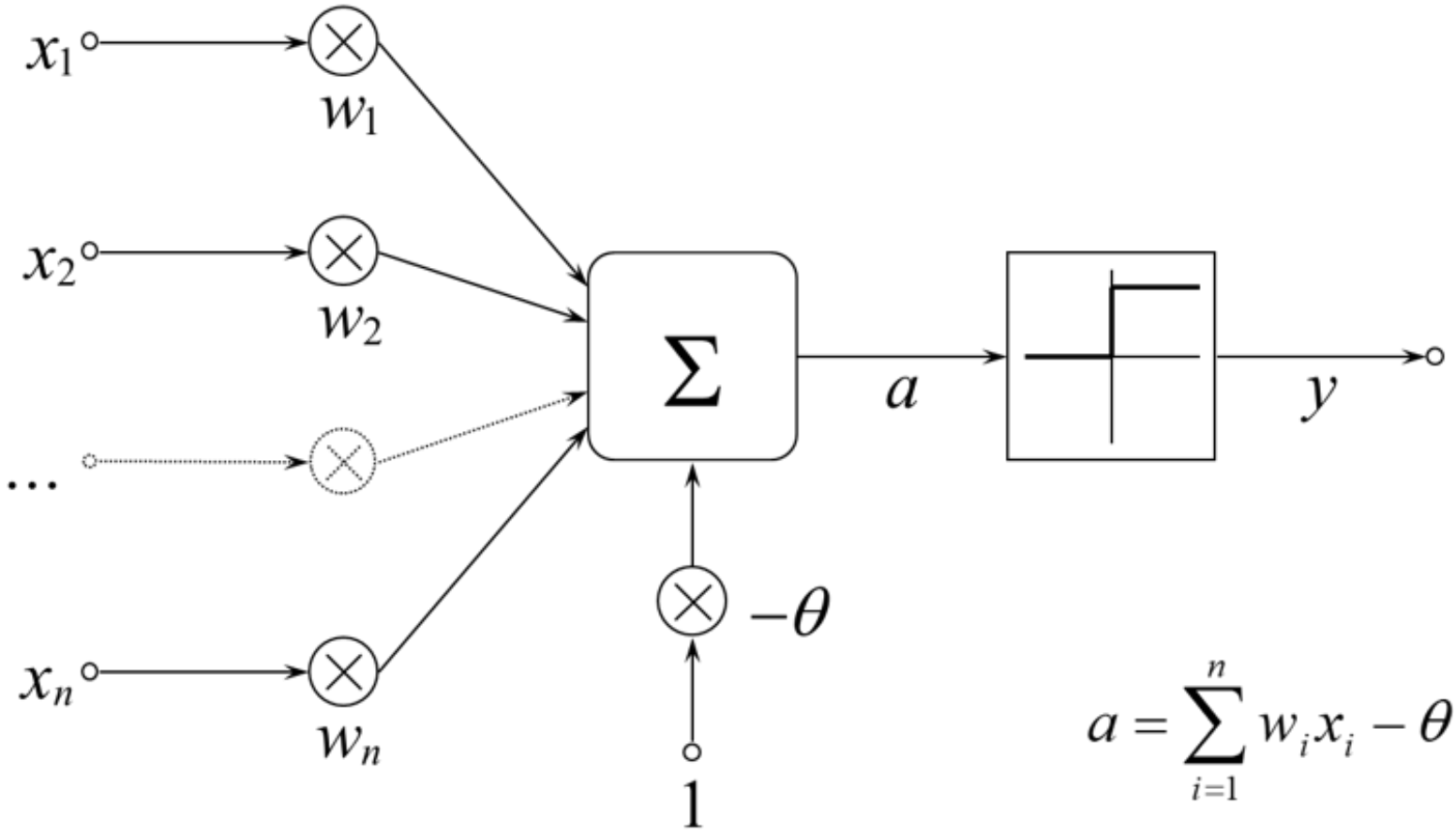
Mc-Culloch and Pitts Neuron

- The McCulloch-Pitts neuron, also known as the binary threshold neuron, is a simple mathematical model of a biological neuron that was introduced by Warren McCulloch and Walter Pitts in 1943. It is one of the earliest and most fundamental models of artificial neural networks.
- The McCulloch-Pitts neuron is a computational unit that takes in one or more binary inputs, sums them up, and applies a binary threshold function to produce a binary output. The threshold function is such that if the weighted sum of the inputs is greater than a certain threshold value, the neuron outputs a 1 (i.e., it fires), and otherwise, it outputs a 0 (i.e., it remains inactive).

Mc-Culloch and Pitts Neuron

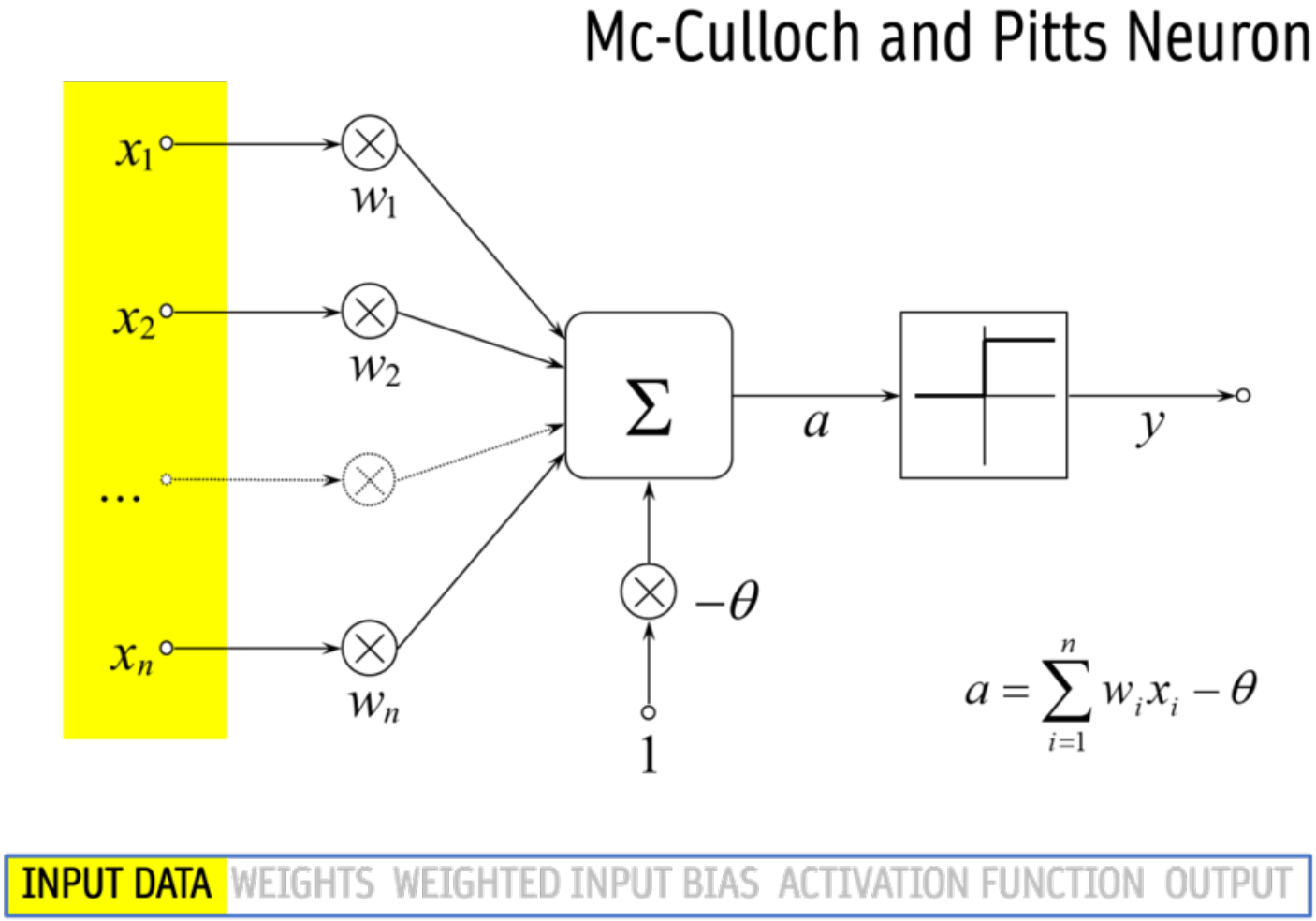
- The McCulloch-Pitts neuron can be used to model basic logic operations, such as AND, OR, and NOT, and can be combined in various ways to build more complex computational systems. While the McCulloch-Pitts neuron is a highly simplified model of a biological neuron, it serves as a foundation for more sophisticated neural network models, such as the perceptron, which was introduced a few years later by Frank Rosenblatt.

Mc-Culloch and Pitts Neuron

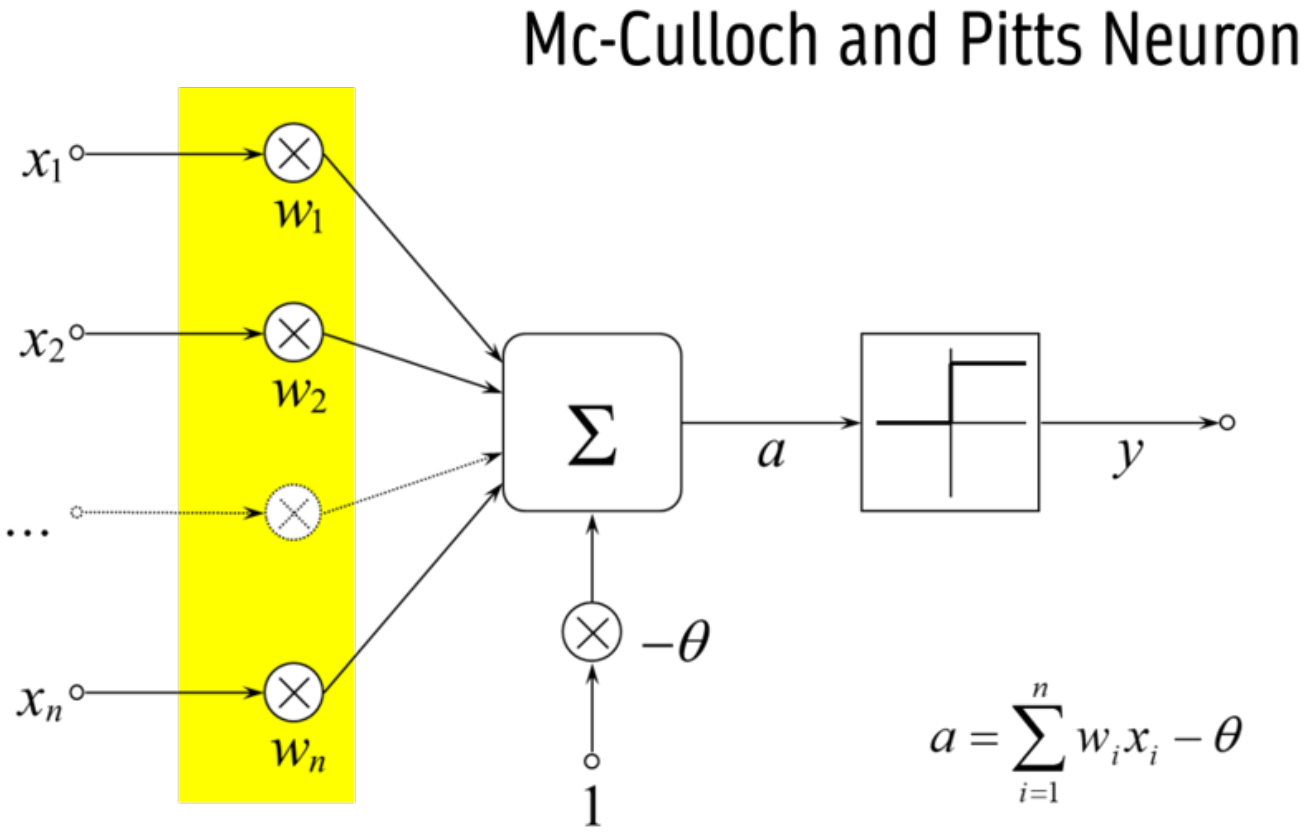


$$a = \sum_{i=1}^{n} w_i x_i - \theta$$

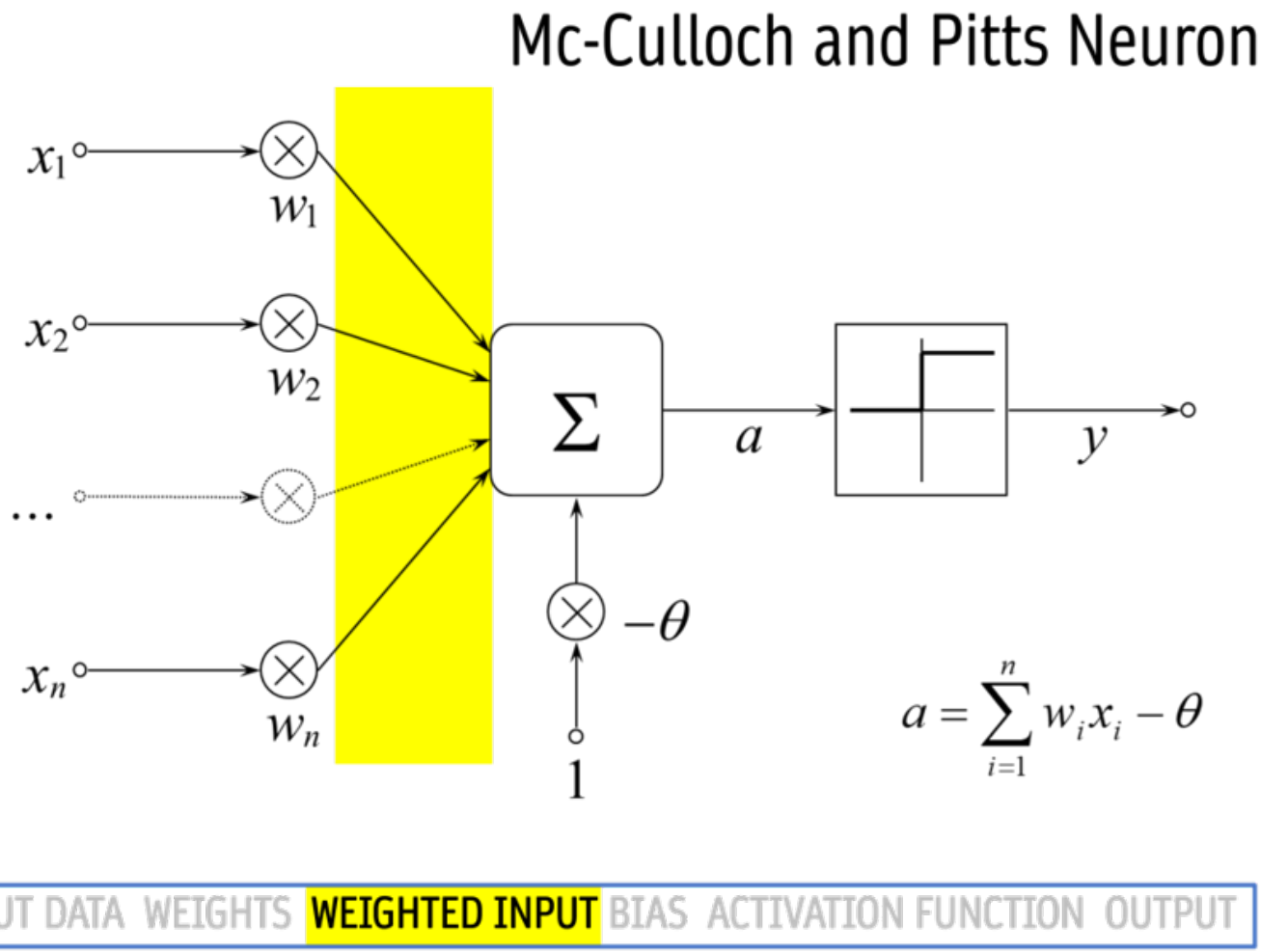
NN Data Flow


Mc-Culloch and Pitts Neuron
x_1
x_2
...
x_n
w_1
w_2
w_n
Σ
a
y
$-\theta$
1
$a = \sum_{i=1}^{n} w_i x_i - \theta$
INPUT DATA
WEIGHTS
WEIGHTED INPUT
BIAS
ACTIVATION FUNCTION
OUTPUT

NN Data Flow


Mc-Culloch and Pitts Neuron
x_1
x_2
...
x_n
w_1
w_2
w_n
Σ
a
y
$-\theta$
1
$a = \sum_{i=1}^{n} w_i x_i - \theta$
INPUT DATA
WEIGHTS
WEIGHTED INPUT
BIAS
ACTIVATION FUNCTION
OUTPUT

NN Data Flow

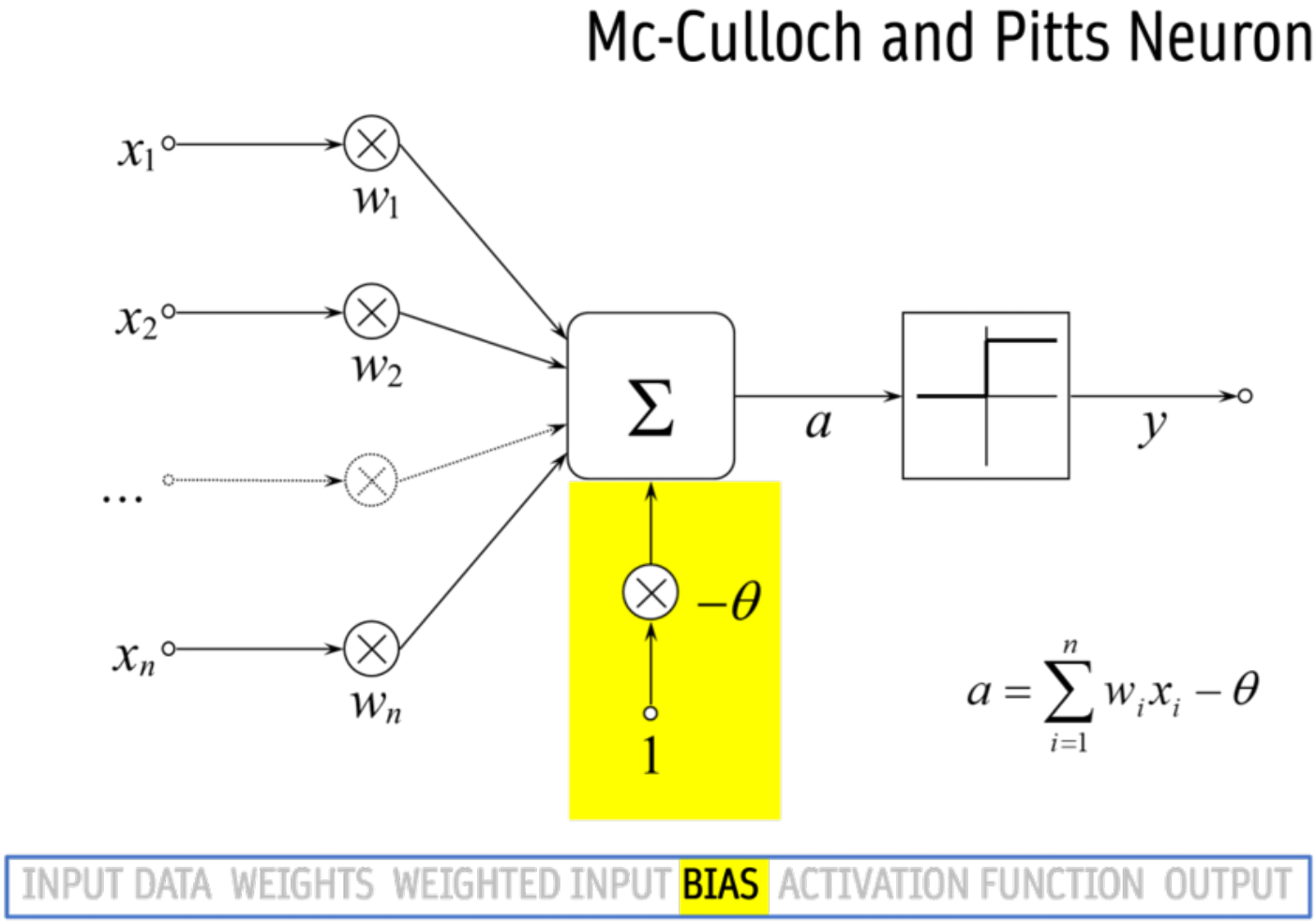

Mc-Culloch and Pitts Neuron
x_1
x_2
...
x_n
w_1
w_2
w_n
Σ
a
y
$-\theta$
1
$a = \sum_{i=1}^{n} w_i x_i - \theta$
INPUT DATA WEIGHTS WEIGHTED INPUT BIAS ACTIVATION FUNCTION OUTPUT

Mc-Culloch and Pitts Neuron

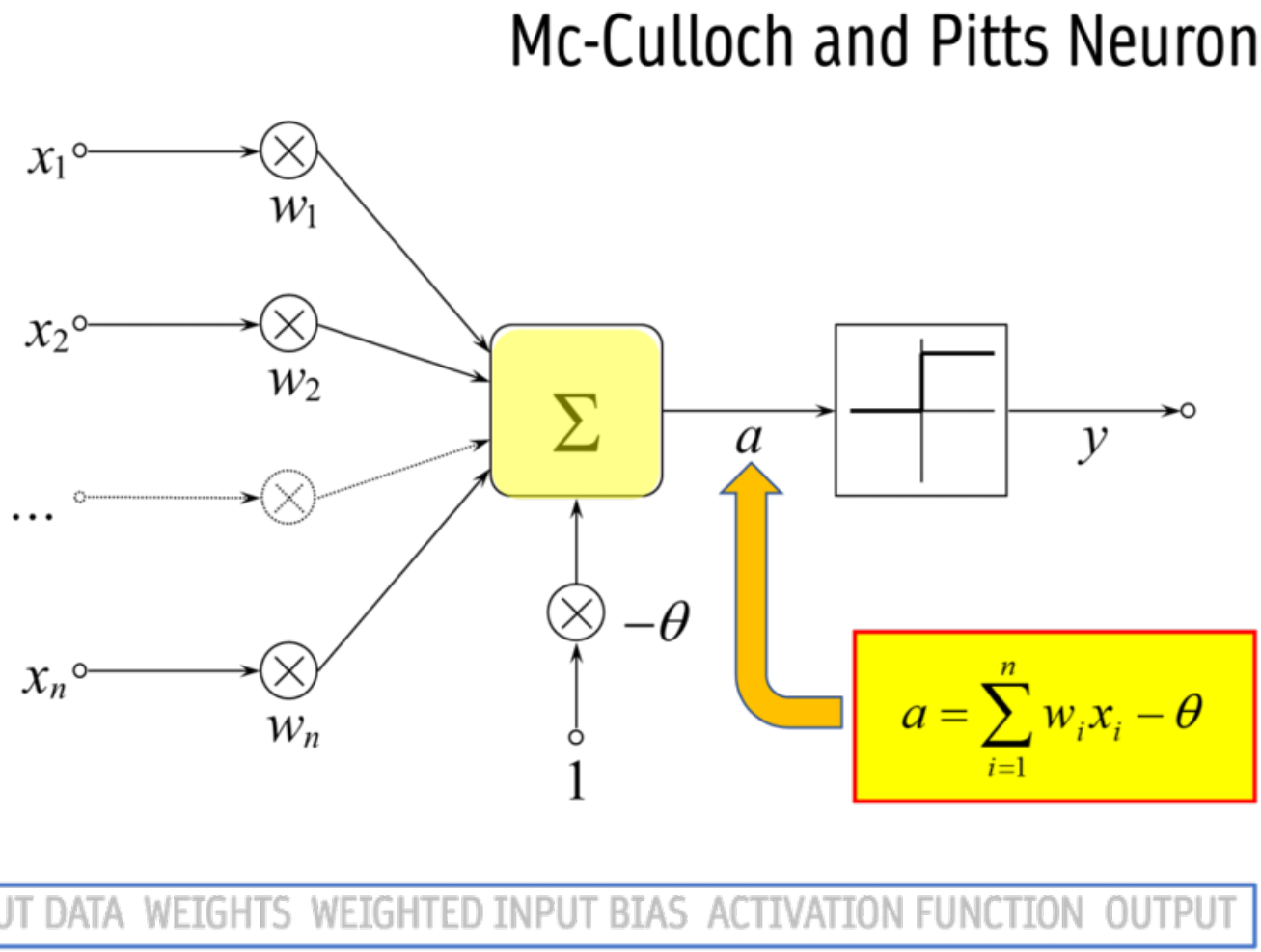
From a functional point of view

- an input signal formally present but associated with a zero weight is equivalent to an absence of signal;
- the threshold can be considered as an additional synapse, connected in input with a fixed weight equal to 1;

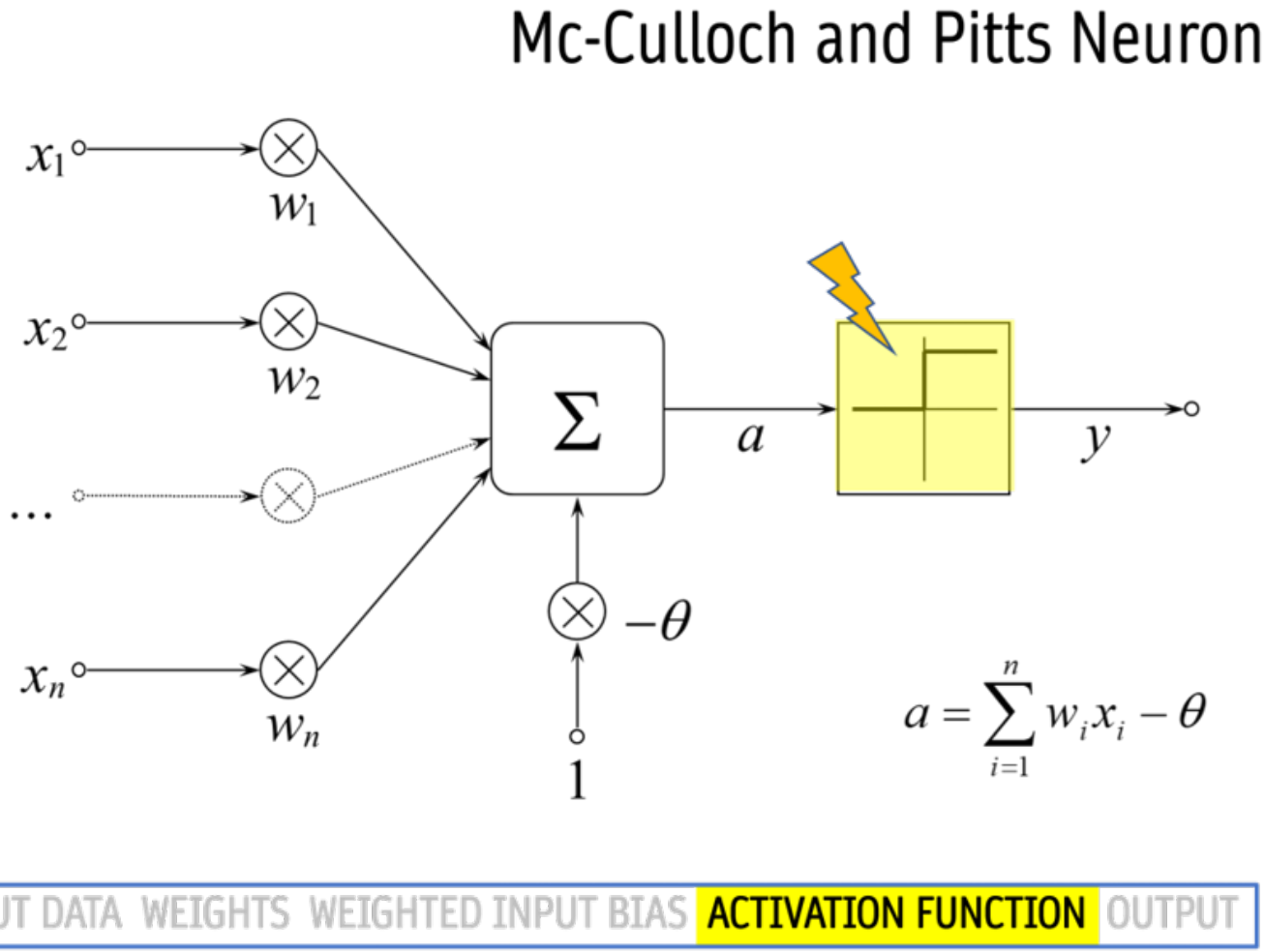
NN Data Flow


Mc-Culloch and Pitts Neuron
x_1
x_2
...
x_n
w_1
w_2
w_n
Σ
a
y
$-\theta$
1
$a = \sum_{i=1}^{n} w_i x_i - \theta$
INPUT DATA WEIGHTS WEIGHTED INPUT BIAS ACTIVATION FUNCTION OUTPUT

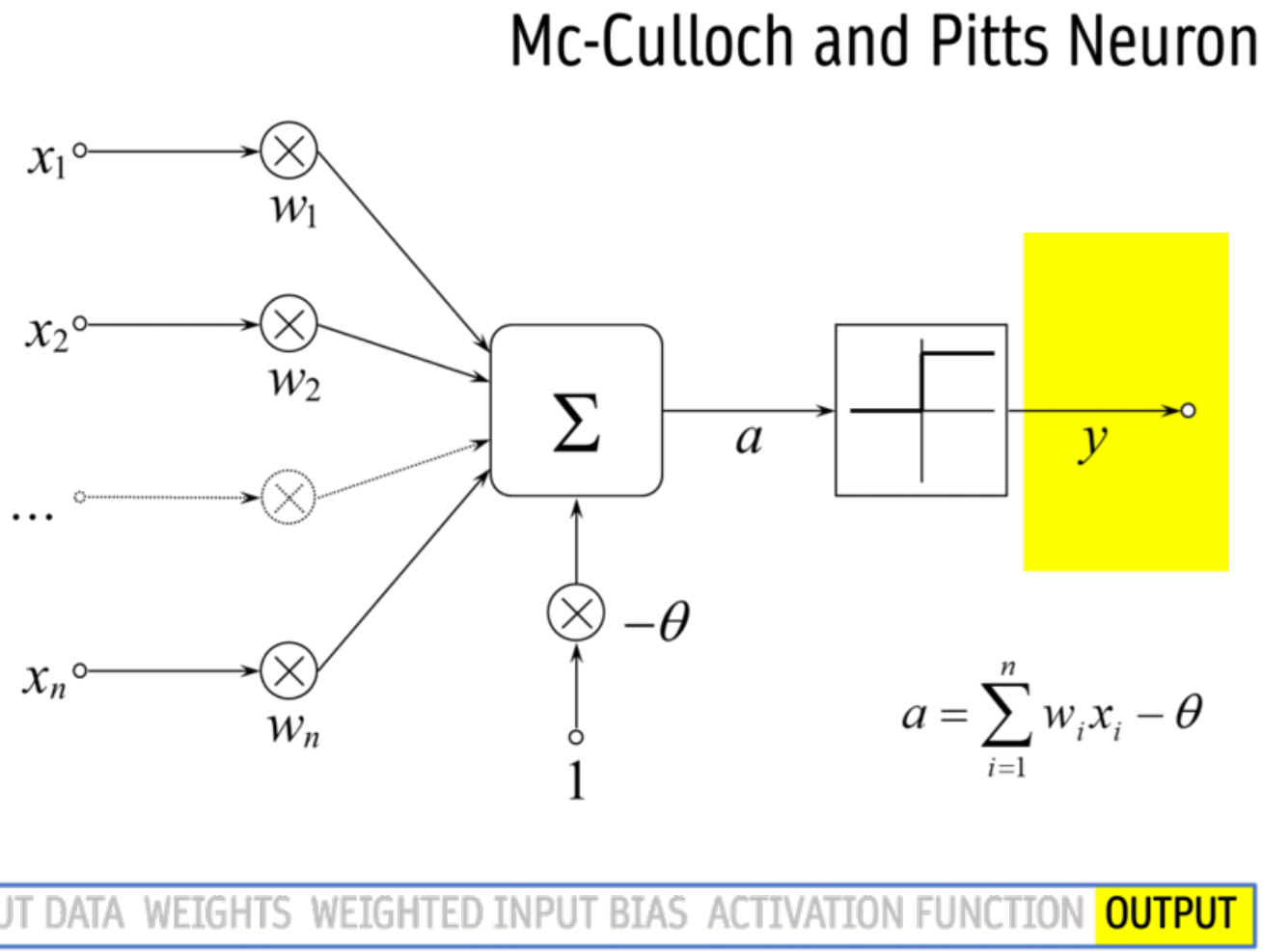
NN Data Flow


Mc-Culloch and Pitts Neuron
x_1
x_2
...
x_n
w_1
w_2
w_n
Σ
a
y
$-\theta$
1
$a = \sum_{i=1}^{n} w_i x_i - \theta$
INPUT DATA WEIGHTS WEIGHTED INPUT BIAS ACTIVATION FUNCTION OUTPUT

NN Data Flow


Mc-Culloch and Pitts Neuron
x_1
x_2
...
x_n
w_1
w_2
w_n
Σ
a
y
$-\theta$
1
$a = \sum_{i=1}^{n} w_i x_i - \theta$
INPUT DATA WEIGHTS WEIGHTED INPUT BIAS ACTIVATION FUNCTION OUTPUT

NN Data Flow


Mc-Culloch and Pitts Neuron
x_1
x_2
...
x_n
w_1
w_2
w_n
Σ
a
y
$-\theta$
1
$a = \sum_{i=1}^{n} w_i x_i - \theta$
INPUT DATA WEIGHTS WEIGHTED INPUT BIAS ACTIVATION FUNCTION OUTPUT

Activation Function

$$a = \sum_{i=1}^{n} w_i \, x_i - \theta \tag{1}$$

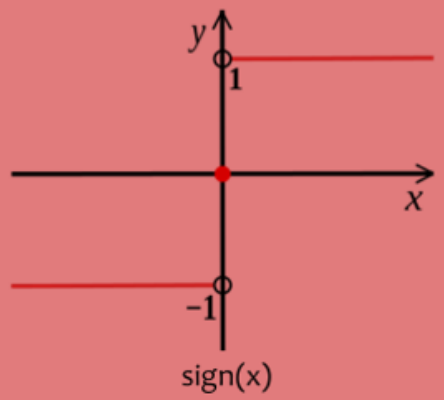
$$y = f(a) = \begin{cases} 0, & \text{if } a \leq 0 \\ 1, & \text{if } a > 0 \end{cases} \tag{2}$$

- The function f is called the response or activation function:
- in the McCulloch and Pitts neuron f is simply the step function, so the answer is binary: it is 1 if the weighted sum of the stimuli exceeds the internal threshold; 0 otherwise.
- Other models of artificial neurons predict continuous response functions

Activation Function

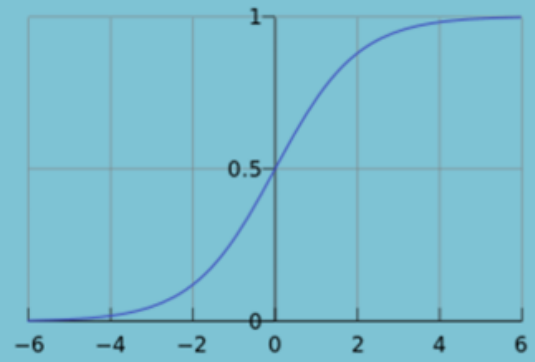
This decision function isn't differentiable:

$$h(\mathbf{x}) = \text{sign}(\boldsymbol{\theta}^T \mathbf{x})$$



Use a differentiable function instead:

$$p_{\boldsymbol{\theta}}(y = 1|\mathbf{x}) = \frac{1}{1 + \exp(-\boldsymbol{\theta}^T \mathbf{x})}$$

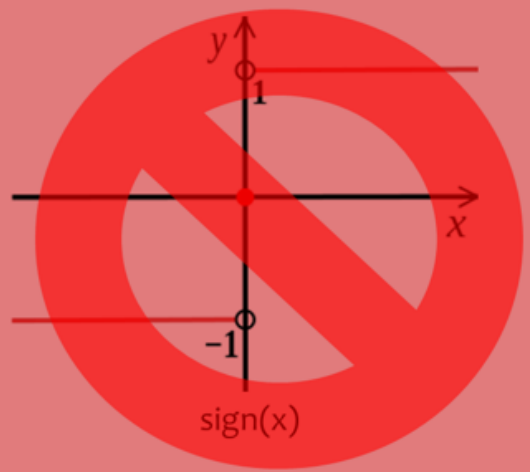


$$\text{logistic}(u) \equiv \frac{1}{1+e^{-u}}$$

Activation Function

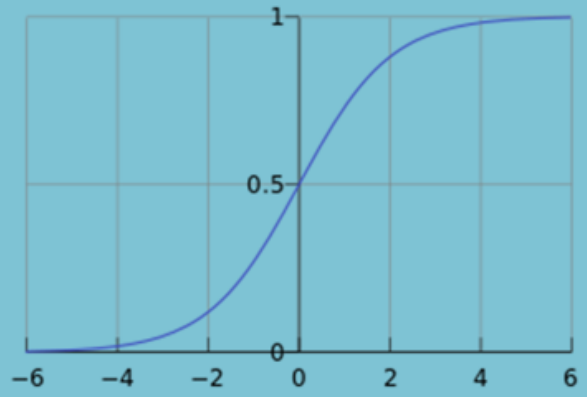
This decision function isn't differentiable:

$$h(\mathbf{x}) = \text{sign}(\boldsymbol{\theta}^T \mathbf{x})$$



Use a differentiable function instead:

$$p_{\boldsymbol{\theta}}(y = 1|\mathbf{x}) = \frac{1}{1 + \exp(-\boldsymbol{\theta}^T \mathbf{x})}$$



$$\text{logistic}(u) \equiv \frac{1}{1+e^{-u}}$$

Subsection 2

The Feed-Forward Neural Network

Neural Network Basic Constituents

More generally, a neural network consists of:

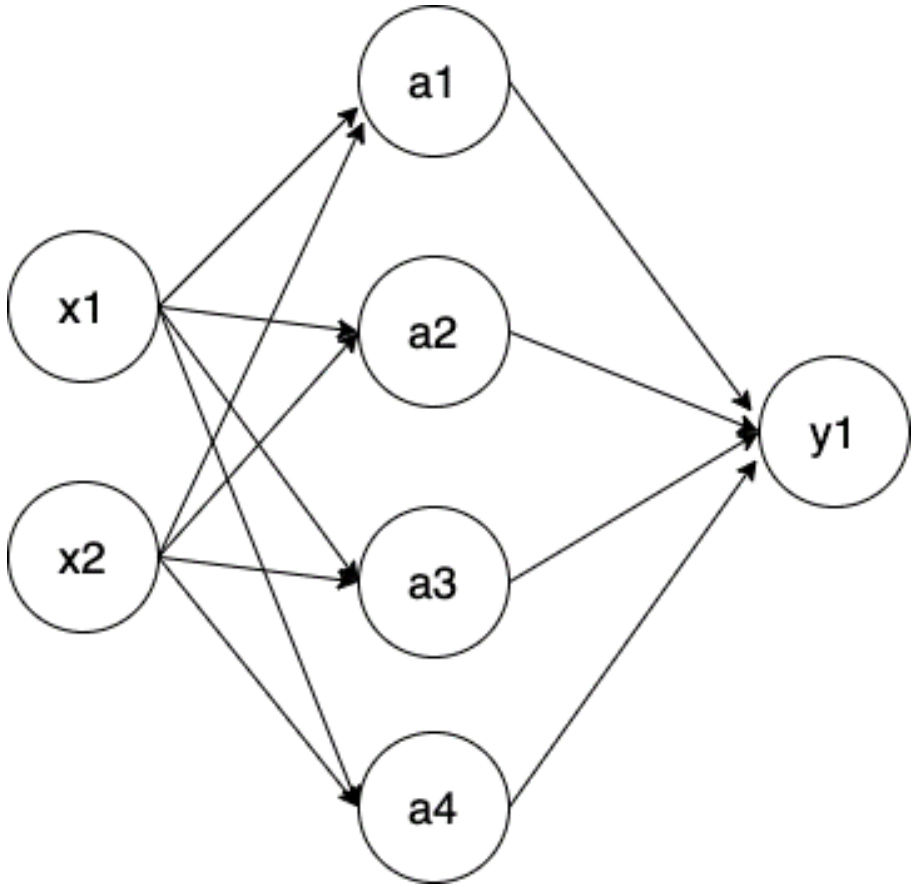
- A set of nodes (neurons), or units connected by links.
- A set of **weights** associated with links.
- A set of thresholds or activation levels.

Neural network design requires:

- 1. The choice of the number and type of units.
- 2. The determination of the morphological structure.
- 3. Coding of training examples, in terms of network inputs and outputs.
- 4. Initialization and training of weights on interconnections, through the set of learning examples.

A simple neural networw

- Feedforward neural networks are what we've primarily been focusing on within this section.
- They are comprised of an input layer, a hidden layer or layers, and an output layer.
- Data is fed into these models through the input layer.
- In this simple example we suppose to have to classify input data into one of two possible categories (binary classification).

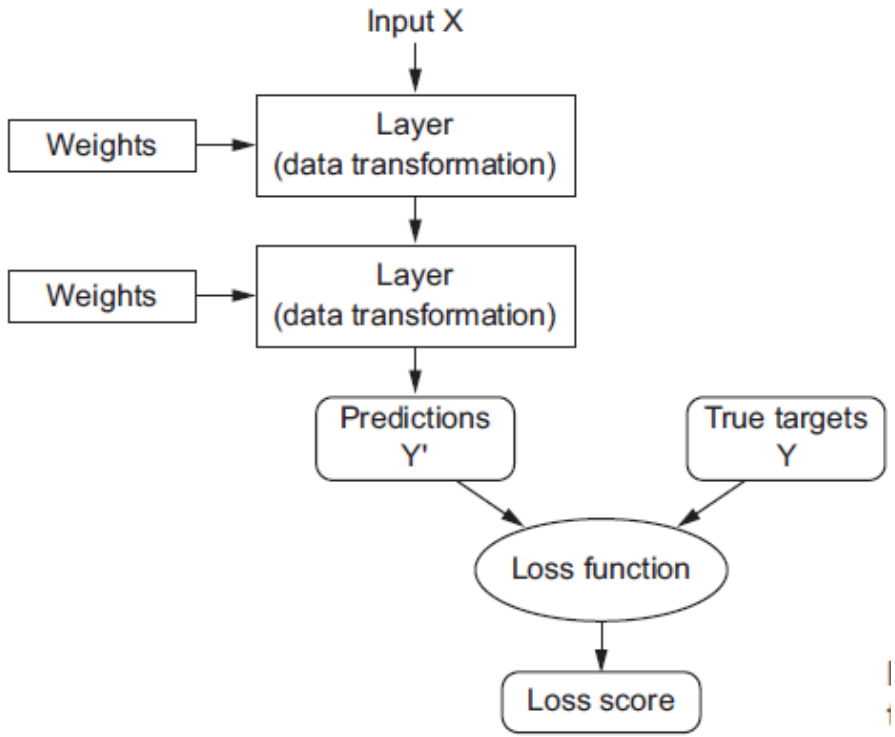


Neural Network Basic Constituents

- The specification of what a layer does to its input data is stored in the layer's weights, which in essence are a bunch of numbers.
- In technical terms, we could say that the transformation implemented by a layer is parameterized by its weights (Weights are also sometimes called the parameters of a layer.)
- In this context, **learning means finding a set of values for the weights of all layers in a network, such that the network will correctly map example inputs to their associated targets**.

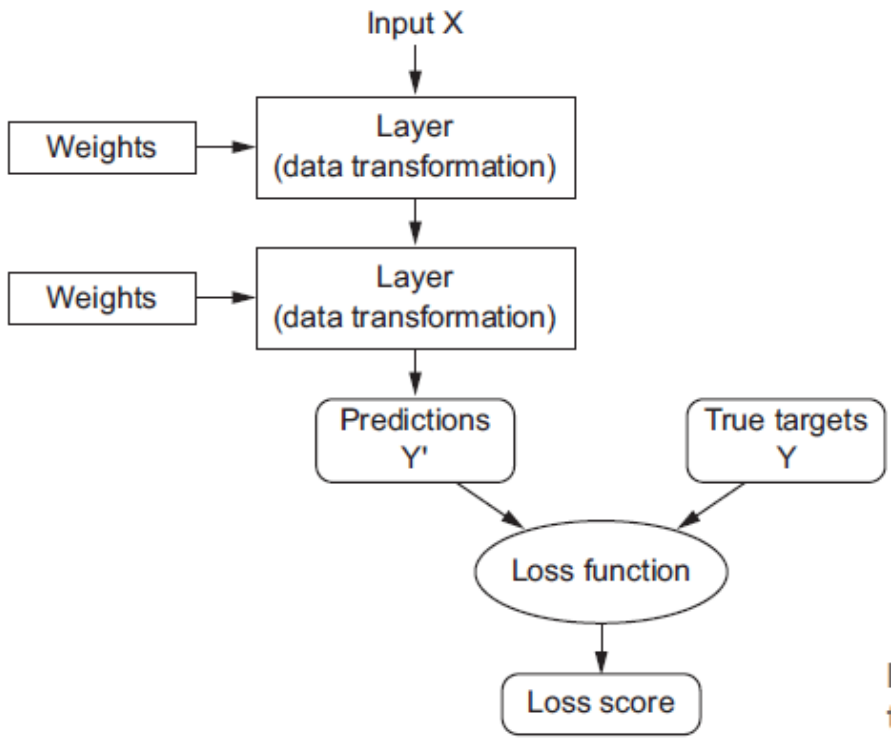
Loss Function

- To control the output of a neural network, you need to be able to measure how far this output is from what you expected.
- This is the job of the **loss function** of the network, also called the objective function.



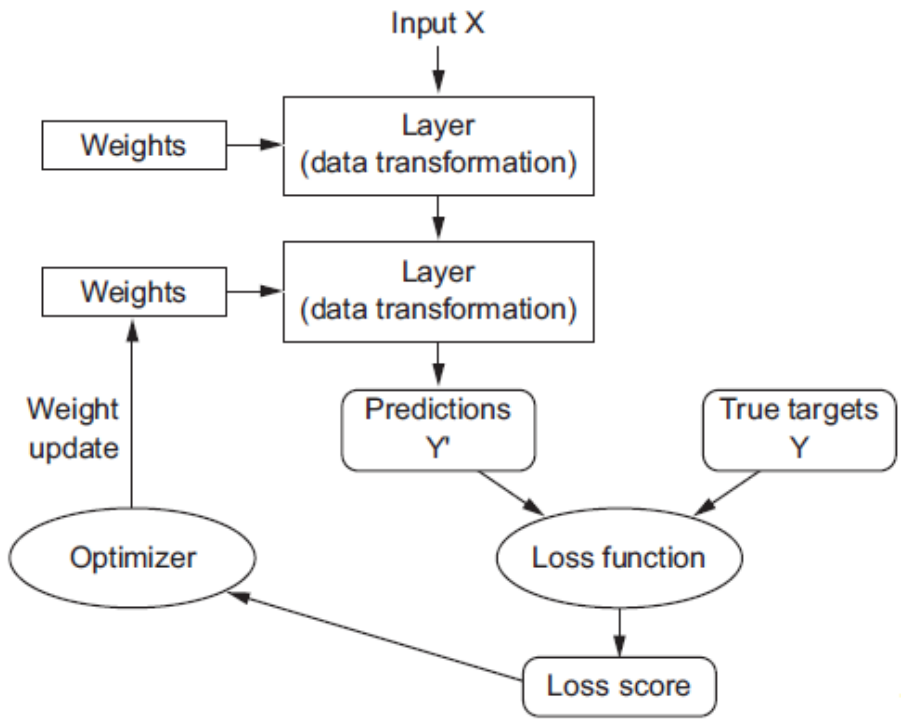
Loss Function

- The loss function takes the predictions of the network and the true target (what you wanted the network to output) and **computes a distance score, capturing how well the network has done on this specific example**

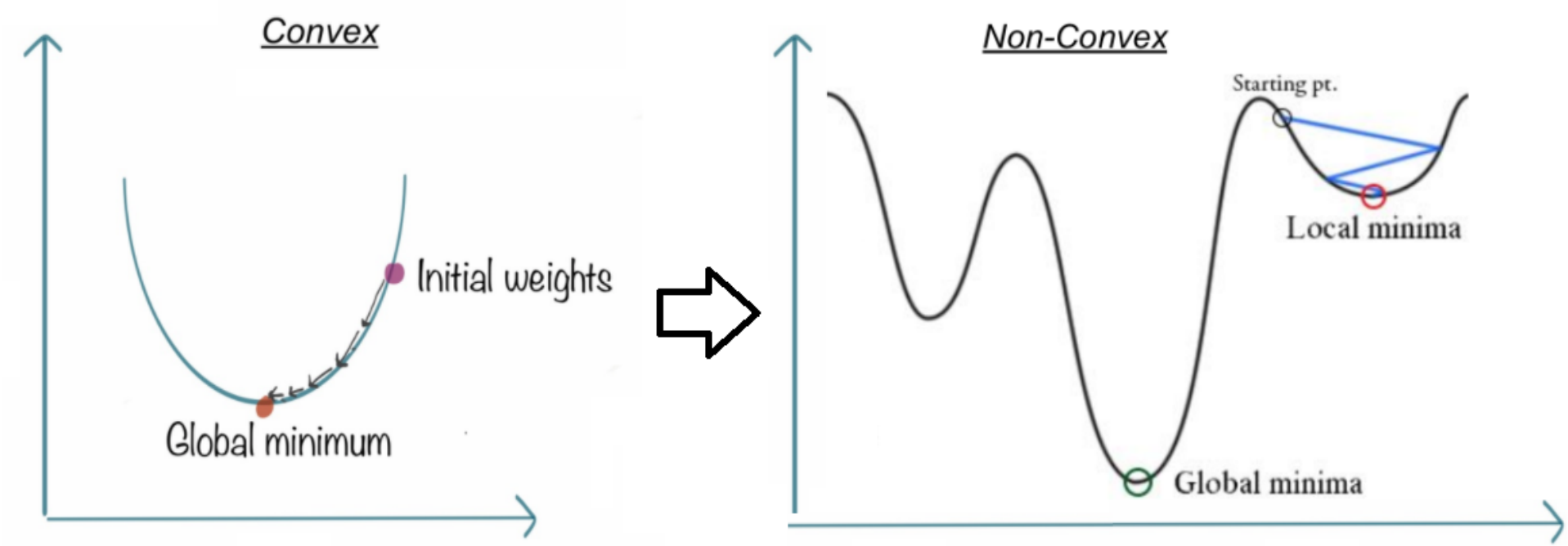


Backpropagation

- The fundamental trick in deep learning is to use this score as a feedback signal to adjust the value of the weights a little, in a direction that will lower the loss score for the current example.
- This adjustment is the job of the optimizer, which implements what is called the Backpropagation algorithm: the central algorithm in deep learning.



Loss Function



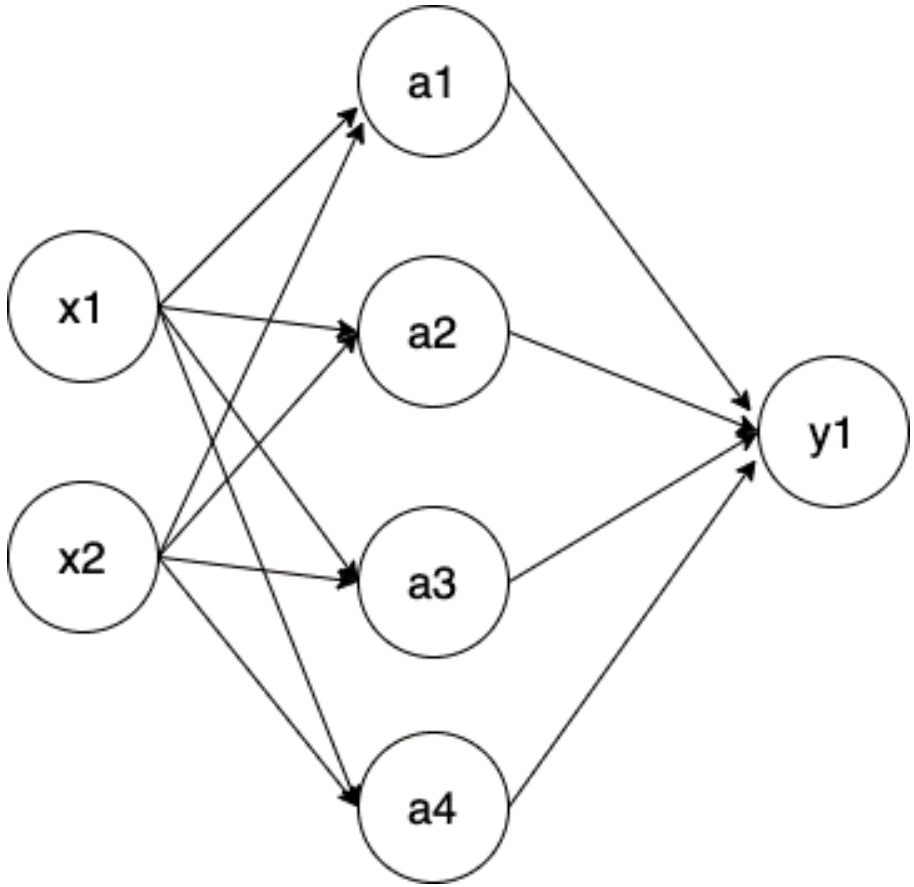
Subsection 3

Implementing a Single Layer NN

Implementing a Single Layer NN

- In each hidden unit, take a_1 as example, **a linear operation followed by an activation function, f, is performed**.
- So given input $x = (x_1, x_2)$, **inside node** a_1, we have:

$$z_1 = w_{11}x_1 + w_{12}x_2 + b_1$$
$$a_1 = f(w_{11}x_1 + w_{12}x_2 + b_1) = f(z_1)$$

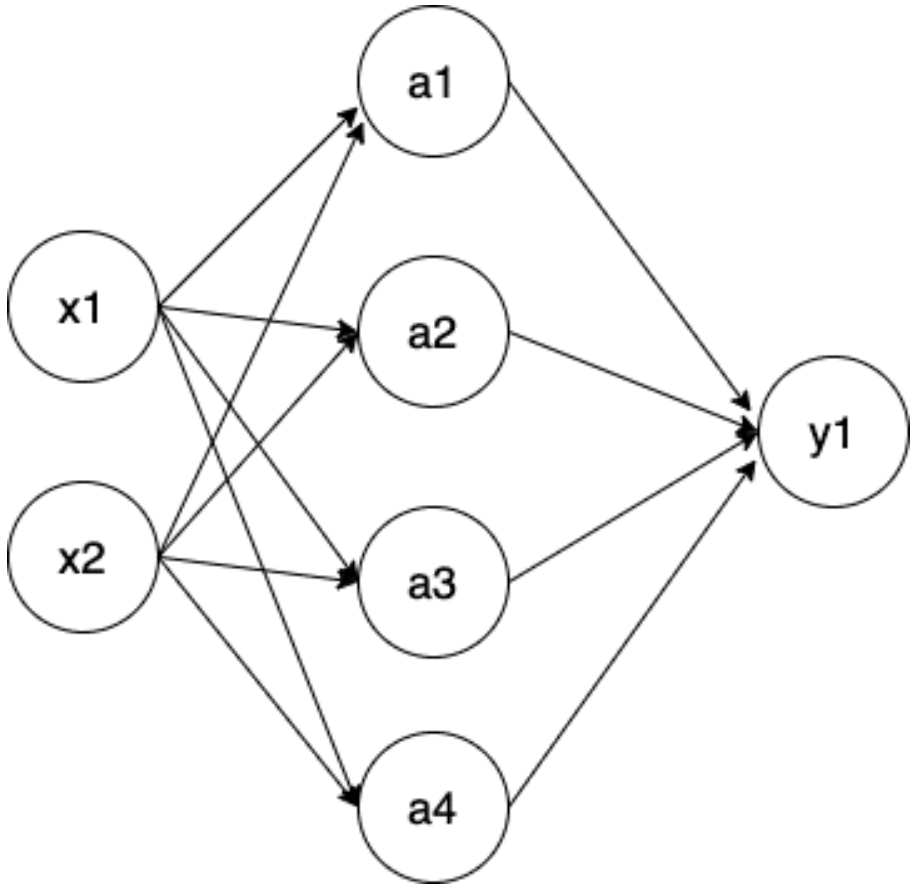


Implementing a Single Layer NN

- Same for node a_2, it would have:

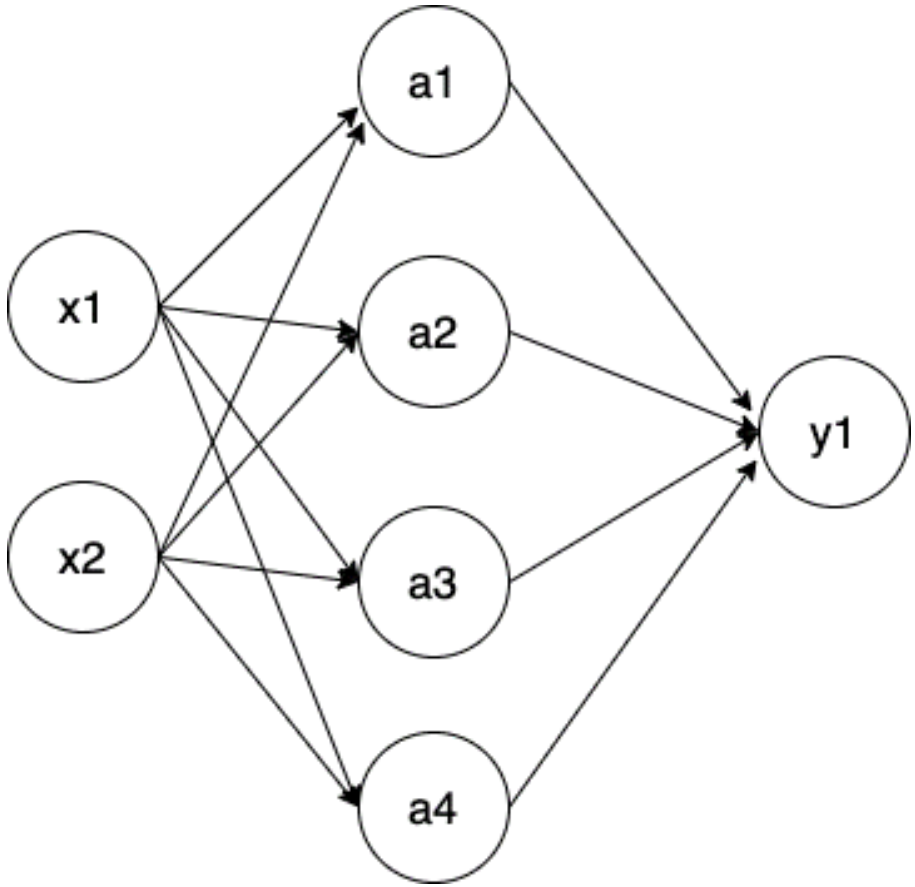
$$z_2 = w_{21}x_1 + w_{22}x_2 + b_2$$
$$a_2 = f(w_{21}x_1 + w_{22}x_2 + b_2) = f(z_2)$$

- And same for a_3 and a_4 and so on



Implementing a Single Layer NN

- The output is one single value y_1 in $[0, 1]$.
- We can think of this simple example as a binary classification task with a prediction of probability



Implementing a single Layer NN

We can also write in a more compact form the computation of the pre-activation function performed by the hidden layer:

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \Rightarrow Z^{[1]} = W^{[1]} \cdot X + B^{[1]} \quad (3)$$

Implementing a single Layer NN

Let's assume that the first activation function is the tanh and the output activation function is the *sigmoid*. So the result of the hidden layer is:

$$A^{[1]} = \tanh Z^{[1]}$$

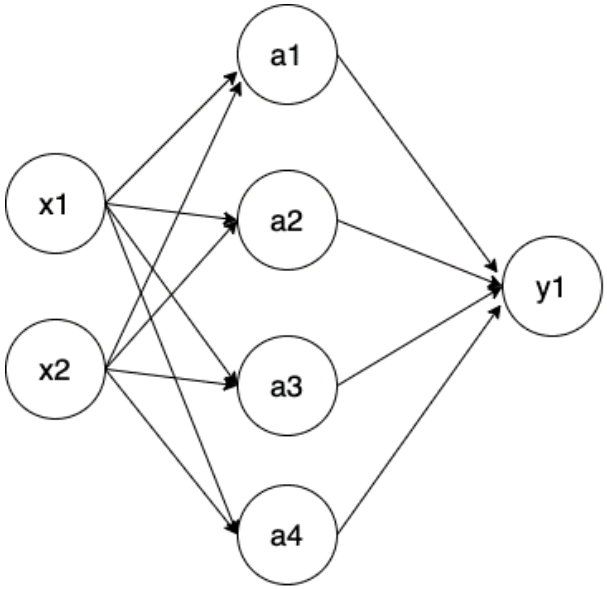
This result is applied to the output node which will perform another linear operation with a different set of weights, $W^{[2]}$:

$$Z^{[2]} = W^{[2]} \cdot A^{[1]} + B^{[2]}$$

and the final output will be the result of the application of the output node activation function (the sigmoid) to this value:

$$\hat{y} = \sigma(Z^{[2]}) = A^{[2]}$$

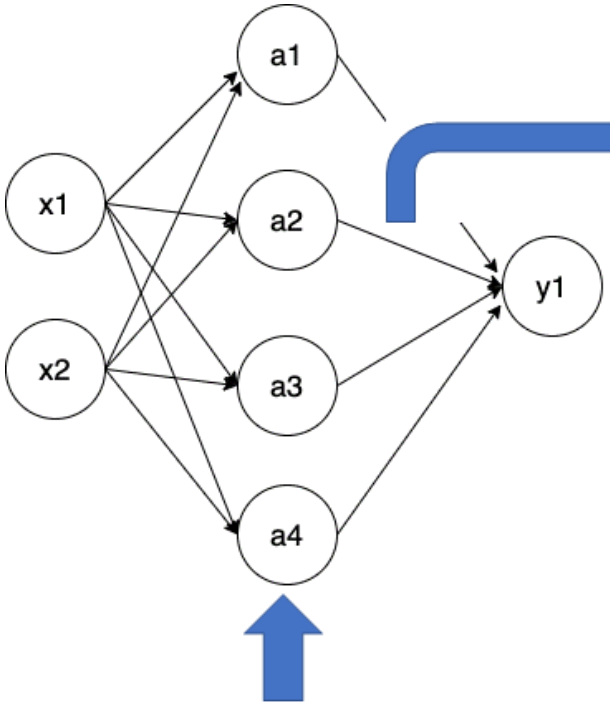
Implementing a single Layer NN



This is the first part of the computation performed by the nodes of the inner layer ...

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \Rightarrow Z^{[1]} = W^{[1]} \cdot X + B^{[1]}$$

Implementing a single Layer NN

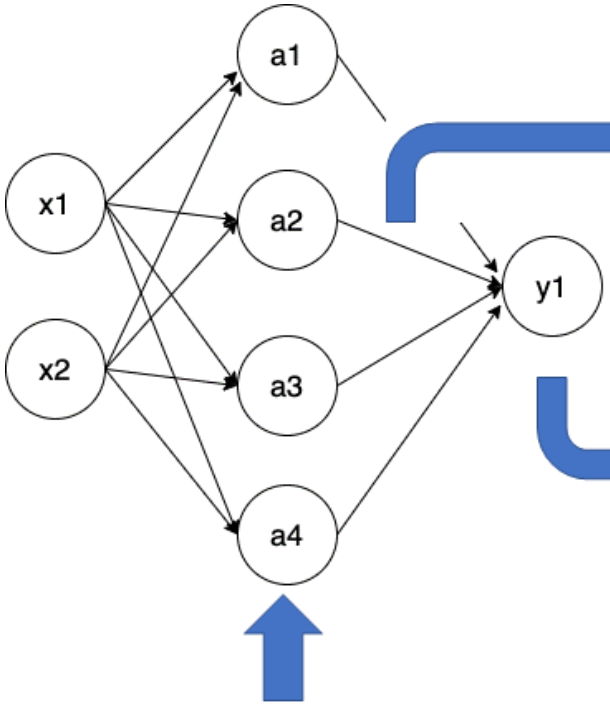


Let's assume that the activation function of the hidden layer is the tanh, so the total result of the hidden layer is

$$A^{[1]} = \tanh Z^{[1]}$$

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \Rightarrow Z^{[1]} = W^{[1]} \cdot X + B^{[1]}$$

Implementing a single Layer NN



Let's assume that the activation function of the hidden layer is the tanh, so the total result of the hidden layer is

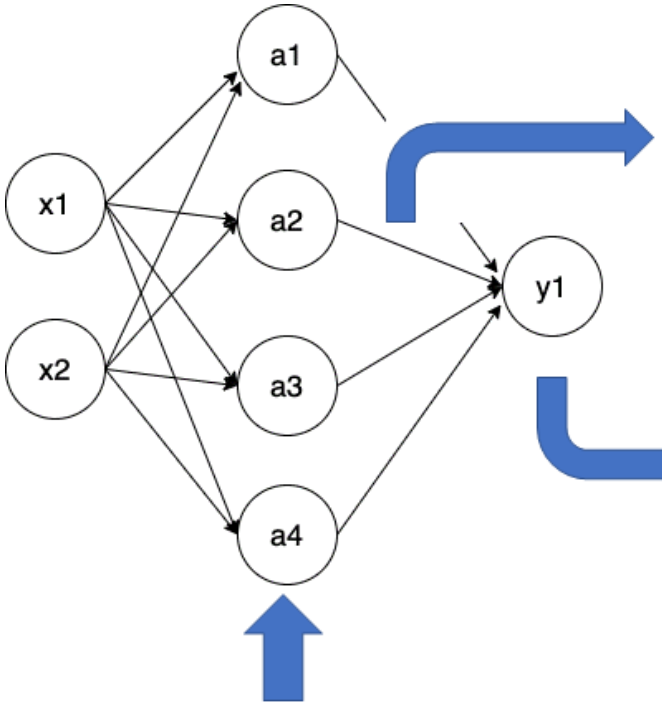
$$A^{[1]} = \tanh Z^{[1]}$$

The previous result is applied to the output node which will perform another linear operation with a **DIFFERENT** set of weights

$$Z^{[2]} = W^{[2]} \cdot A^{[1]} + B^{[2]}$$

$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \Rightarrow Z^{[1]} = W^{[1]} \cdot X + B^{[1]}$$

Implementing a single Layer NN



$$\begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \Rightarrow Z^{[1]} = W^{[1]} \cdot X + B^{[1]}$$

Let's assume that the activation function of the hidden layer is the tanh, so the total result of the hidden layer is

$$A^{[1]} = \tanh Z^{[1]}$$

The previous result is applied to the output node which will perform another linear operation with a **DIFFERENT** set of weights

$$Z^{[2]} = W^{[2]} \cdot A^{[1]} + B^{[2]}$$

The Final Output will be the result of the application of the output node activation function (sigmoid):

$$\hat{y} = \sigma(Z^{[2]}) = A^{[2]}$$

Weights Initialization

- Our neural network has 1 hidden layer and 2 layers in total (hidden layer + output layer), so there are 4 weight matrices to initialize ($W^{[1]}, b^{[1]}$ and $W^{[2]}, b^{[2]}$).
- Notice that the weights are initialized relatively small so that the gradients would be higher thus learning faster in the beginning phase.

```
def init_weights(n_input, n_hidden, n_output):
    params = {}
    params['W1'] = np.random.randn(n_hidden, n_input) * 0.01
    params['b1'] = np.zeros((n_hidden, 1))
    params['W2'] = np.random.randn(n_output, n_hidden) * 0.01
    params['b2'] = np.zeros((n_output, 1))

    return params
```

Weights Initialization

- Our neural network has 1 hidden layer and 2 layers in total (hidden layer + output layer), so there are 4 weight matrices to initialize ($W^{[1]}, b^{[1]}$ and $W^{[2]}, b^{[2]}$).
- Notice that the weights are initialized relatively small so that the gradients would be higher thus learning faster in the beginning phase.

```
params = init_weights(20, 10, 1)

print('W1 shape', params['W1'].shape)
print('b1 shape', params['b1'].shape)
print('W2 shape', params['W2'].shape)
print('b2 shape', params['b2'].shape)
```

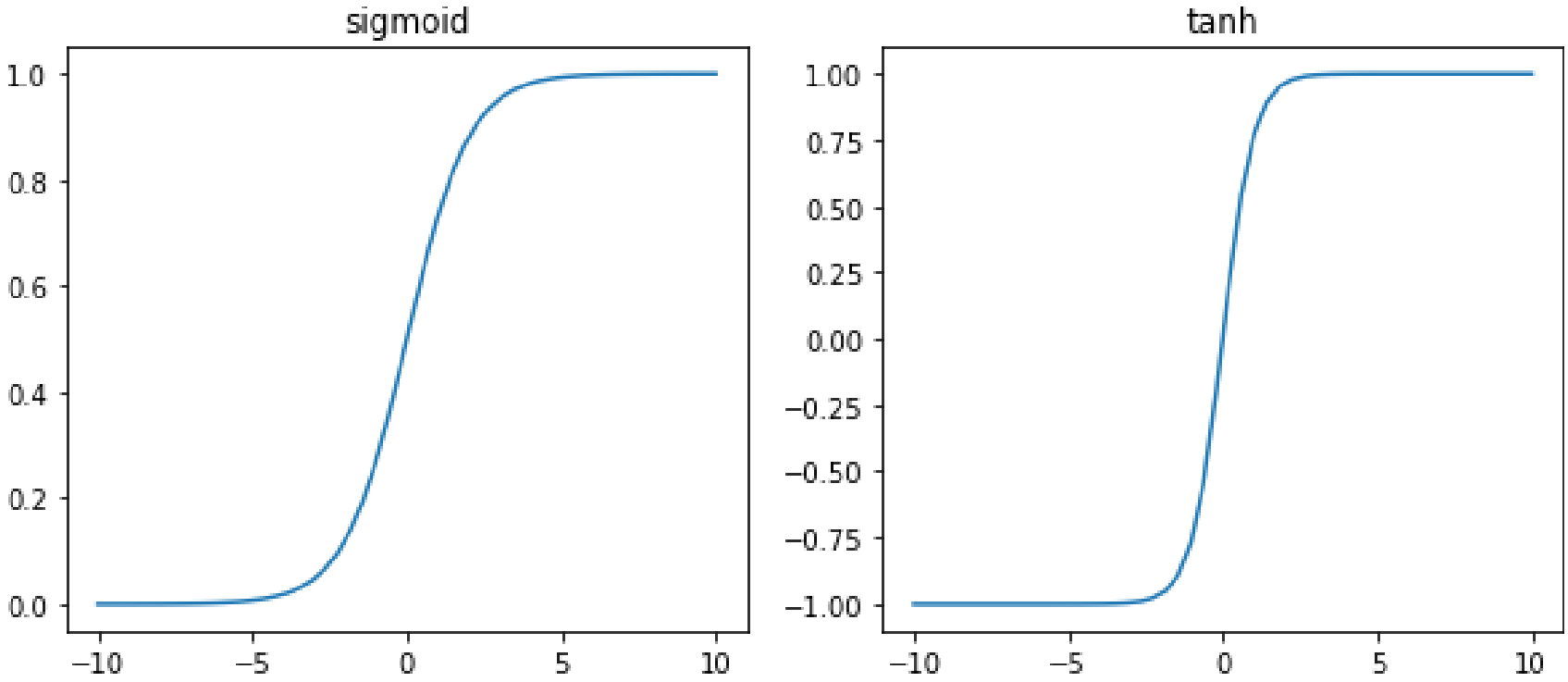
Forward Propagation

$$\begin{aligned}\hat{y} = A^{[2]} &= \sigma\left(Z^{[2]}\right) = \sigma\left(W^{[2]} \cdot A^{[1]} + B^{[2]}\right) \\ &= \sigma\left(W^{[2]} \cdot \tanh Z^{[1]} + B^{[2]}\right) \\ &= \sigma\left[W^{[2]} \cdot \tanh\left(W^{[1]} \cdot X + B^{[1]}\right) + B^{[2]}\right]\end{aligned}$$

```
def forward(X, params):
    W1, b1, W2, b2 = params['W1'], params['b1'], params['W2'], params['b2']
    A0 = X
    cache = {}
    Z1 = np.dot(W1, A0) + b1
    A1 = tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)
    cache['Z1'] = Z1
    cache['A1'] = A1
    cache['Z2'] = Z2
    cache['A2'] = A2
    return  cache
```

Activation Functions

Function tanh and sigmoid looks as below. Notice that the only difference of these functions is the scale of y.



Logistic Loss Function

Since we have a binary classification problem, we can assume a Logistic Loss Function (see the problem of logistic regression)

$$L(y, \hat{y}) = \begin{cases} -\log \hat{y} & \text{when } y = 1 \\ -\log(1 - \hat{y}) & \text{when } y = 0 \end{cases} \tag{4}$$

$$L(y, \hat{y}) = -[y \log \hat{y} + (1 - y) \log (1 - \hat{y})]$$

Where $\hat{y}$ is our prediction ranging in $[0, 1]$ and y is the true value.

Logistic Loss Function

$$L(y, \hat{y}) = -[y \log \hat{y} + (1 - y) \log (1 - \hat{y})]$$

```python
def loss(Y, Y_hat):
    """
    Y: vector of true value
    Y_hat: vector of predicted value
    """
    assert Y.shape[0] == 1
    assert Y.shape == Y_hat.shape
    m = Y.shape[1]
    s = Y * np.log(Y_hat) + (1 - Y) * np.log(1 - Y_hat)
    loss = -np.sum(s) / m
    return loss
```

Delta Rule

- Given a generic actual value y, we want to minimize the loss L, and the technic we are going to apply here is gradient descent;
- basically what we need to do is to apply derivative to our variables and move them slightly down to the optimum.
- Here we have 2 variables, W and b, and for this example, the update formula of them would be:

$$W_{new} = W_{old} - \frac{\partial L}{\partial W} \Rightarrow \Delta W = -\frac{\partial L}{\partial W}$$

$$b_{new} = b_{old} - \frac{\partial L}{\partial b} \Rightarrow \Delta b = -\frac{\partial L}{\partial b}$$

Delta Rule

- The delta rule algorithm works by computing the gradient of the loss function with respect to each weight.
- Remember that

$$\begin{aligned}\hat{y} = A^{[2]} &= \sigma\left(Z^{[2]}\right) = \sigma\left(W^{[2]} \cdot A^{[1]} + B^{[2]}\right) \\ &= \sigma\left(W^{[2]} \cdot \tanh Z^{[1]} + B^{[2]}\right) \\ &= \sigma\left[W^{[2]} \cdot \tanh\left(W^{[1]} \cdot X + B^{[1]}\right) + B^{[2]}\right]\end{aligned}$$

As you can see $\hat{y}$ depends on both $W^{[1]}$ and $W^{[2]}$. The specification of what a layer does to its input data is stored in the layer's weights. Remember once again that **learning means finding a set of values for the weights of all layers in a network, such that the network will correctly map example inputs to their associated targets**

Delta Rule

- In order to get the derivative of our targets, chain rules would be applied:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial Z} \frac{\partial Z}{\partial W}$$
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial Z} \frac{\partial Z}{\partial b}$$

- Let's focus only on the calculation of the derivative with respect to W since the calculation of the other derivative (with respect to b) is completely equivalent...

Delta Rule

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial Z} \frac{\partial Z}{\partial W}$$

The derivative of the Loss Function with respect to $\hat{y}$ is very easy and can be calculated once for all because it does not depend on the particular layer:

$$L(y, \hat{y}) = -[y \log \hat{y} + (1 - y) \log (1 - \hat{y})]$$

$$\frac{\partial L}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}} = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})} \Rightarrow$$

$$\frac{\partial L}{\partial A^{[2]}} = \frac{A^{[2]} - y}{A^{[2]}(1 - A^{[2]})}$$

Gradient Calculation

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial Z} \frac{\partial Z}{\partial W}$$

Hidden Layer Activation Function (Hyperbolic Tangent)

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}} \Rightarrow \frac{d}{dx} \tanh x = 1 - (\tanh x)^2 \tag{5}$$

Output Layer Activation Function (Sigmoid Function)

$$\sigma(x) = \left[\frac{1}{1 + e^{-x}} \right] \Rightarrow \frac{d}{dx} \sigma(x) = \sigma(x) \cdot (1 - \sigma(x)) \tag{6}$$

Gradient Calculation: $W^{[2]}$ Update

$$\frac{\partial L}{\partial A^{[2]}} = \frac{A^{[2]} - y}{A^{[2]}(1 - A^{[2]})}$$

$$\frac{\partial A^{[2]}}{\partial Z^{[2]}} = \frac{\partial \sigma(Z^{[2]})}{\partial Z^{[2]}} = \sigma(Z^{[2]}) \cdot (1 - \sigma(Z^{[2]})) = A^{[2]}(1 - A^{[2]})$$

$$\frac{\partial Z^{[2]}}{\partial W^{[2]}} = A^{[1]}$$

So the complete gradient is:

$$\begin{aligned} \frac{\partial L}{\partial W^{[2]}} &= \frac{A^{[2]} - y}{A^{[2]}(1 - A^{[2]})} \cdot A^{[2]}(1 - A^{[2]}) \cdot {A^{[1]}}^T \\ &= (A^{[2]} - y) \cdot {A^{[1]}}^T \end{aligned}$$

Gradient Calculation: $W^{[1]}$ Update

$$\frac{\partial L}{\partial A^{[2]}} = \frac{A^{[2]} - y}{A^{[2]}(1 - A^{[2]})}$$

$$\frac{\partial A^{[2]}}{\partial Z^{[2]}} = \frac{\partial \sigma(Z^{[2]})}{\partial Z^{[2]}} = \sigma(Z^{[2]}) \cdot (1 - \sigma(Z^{[2]})) = A^{[2]}(1 - A^{[2]})$$

$$\frac{\partial Z^{[2]}}{\partial W^{[1]}} = ?$$

So the complete gradient is:

$$\frac{\partial L}{\partial W^{[1]}} = \frac{A^{[2]} - y}{A^{[2]}(1 - A^{[2]})} \cdot A^{[2]}(1 - A^{[2]}) \cdot \frac{\partial Z^{[2]}}{\partial W^{[1]}}$$

$$= (A^{[2]} - y) \cdot \frac{\partial Z^{[2]}}{\partial W^{[1]}}$$

Gradient Calculation: $W^{[1]}$ Update

Now we have to calculate

$$\frac{\partial Z^{[2]}}{\partial W^{[1]}}$$

Remember that

$$Z^{[2]} = W^{[2]} \cdot tanh\left(W^{[1]} \cdot X + b^{[1]}\right) + b^{[2]}$$

and

$$\frac{\partial Z^{[2]}}{\partial W^{[1]}} = W^{[2]} \cdot \frac{\partial\, tanh(\dots)}{\partial W^{[1]}} \cdot X = W^{[2]} \cdot \left(1 - tanh^2(\dots)\right) \cdot X$$

Gradient Calculation: $W^{[1]}$ Update

Finally

$$\frac{\partial L}{\partial W^{[1]}} = (A^{[2]} - y) \cdot W^{[2]} \cdot X \cdot \left(1 - tanh^2(\dots)\right)$$
$$= (A^{[2]} - y) \cdot W^{[2]} \cdot X \cdot \left(1 - {A^{[1]}}^2\right)$$

Gradient Calculation: $W^{[2]}$ Update

Since

$$\frac{\partial L}{\partial W^{[2]}} = (A^{[2]} - y) \cdot A^{[1]\,T} \tag{7}$$

We have

$$\Delta W^{[2]} = \frac{1}{m}\left[A^{[2]} - Y\right] A^{[1]\,T} = \frac{1}{m}\Delta^{[2]} A^{[1]\,T} \tag{8}$$

$$\Delta b^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True) \tag{9}$$

Where

$$\Delta^{[2]} = A^{[2]} - Y \tag{10}$$

Gradient Calculation: $W^{[1]}$ Update

$$\begin{aligned}\Delta W^{[1]} &= \frac{1}{m}\left[A^{[2]} - Y\right] \cdot X^T \cdot W^{[2]T} \cdot (1 - A^{[1]^2}) \\ &= \frac{1}{m}\Delta^{[2]} \cdot W^{[2]T} \cdot (1 - A^{[1]^2}) \cdot X^T \\ &= \frac{1}{m}\Delta^{[1]} \cdot X^T \end{aligned} \tag{11}$$

$$\Delta b^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True) \tag{12}$$

Where

$$\Delta^{[1]} = \Delta^{[2]} \cdot W^{[2]T} \cdot (1 - A^{[1]^2})$$

Weights Update

```
def backward(params, cache, X, Y):
    m = X.shape[1]
    W1 = params['W1']
    W2 = params['W2']
    A1 = cache['A1']
    A2 = cache['A2']
    DL2 = A2 - Y
    dW2 = (1 / m) * np.dot(DL2, A1.T)
    db2 = (1 / m) * np.sum(DL2, axis=1, keepdims=True)
    DL1 = np.multiply(np.dot(W2.T, DL2), 1 - np.power(A1, 2))
    dW1 = (1 / m) * np.dot(DL1, X.T)
    db1 = (1 / m) * np.sum(DL1, axis=1, keepdims=True)
    grads = {"dW1": dW1,
             "db1": db1,
             "dW2": dW2,
             "db2": db2}

    return grads
```

Batch Training

- In actual training processes, a batch is trained instead of 1 at a time. The change applied in the formula is trivial, we just need to replace the single vector x with a matrix X with size $n \times m$, where n is number of features and m is the the batch size, samples are stacked column wise, and the following result matrix are applied likewise.
- Also the loss function is the same as logistic regression, but for batch training, we'll take the average loss for all training samples.
- The same it's true for all the other calculation, this explain the presence of the factor $1/m$;

Subsection 4

Keras

Keras

Keras is a popular open-source deep learning library that is written in Python. It is designed to be user-friendly, modular, and extensible, which makes it a great tool for beginners and experts alike. Here are some key points about Keras:

- User-friendly API: Keras provides a high-level API that makes it easy to build and train deep learning models. It supports both CPU and GPU computations, and it runs seamlessly on different platforms.
- Modular architecture: Keras is built on a modular architecture, which means that it provides a set of building blocks that you can combine to create complex deep learning models. This allows you to experiment with different architectures and see what works best for your problem.

Keras

- Supports multiple backends: Keras supports multiple backends, including TensorFlow, CNTK, and Theano. This allows you to choose the backend that works best for your needs.
- Pre-built models and layers: Keras provides a set of pre-built models and layers that you can use to jumpstart your deep learning projects. These models and layers are optimized for different tasks, such as image recognition, natural language processing, and time series analysis.
- Easy customization: Keras allows you to customize your models and layers easily. You can add or remove layers, change the hyperparameters, and fine-tune the models to improve their performance.

Keras

- Built-in utilities: Keras provides a set of built-in utilities that make it easy to preprocess your data, visualize your models, and monitor your training progress. These utilities can save you a lot of time and effort when working on deep learning projects.
- Large community: Keras has a large and active community of users and contributors, who share their knowledge and experience through forums, blogs, and tutorials. This means that you can find resources and help easily when you run into issues or have questions.
- Integration with other libraries: Keras integrates seamlessly with other Python libraries, such as NumPy, Pandas, and Scikit-learn. This allows you to use these libraries together with Keras to preprocess your data, analyze your results, and build end-to-end deep learning pipelines.

Introduction to keras: a practical problem

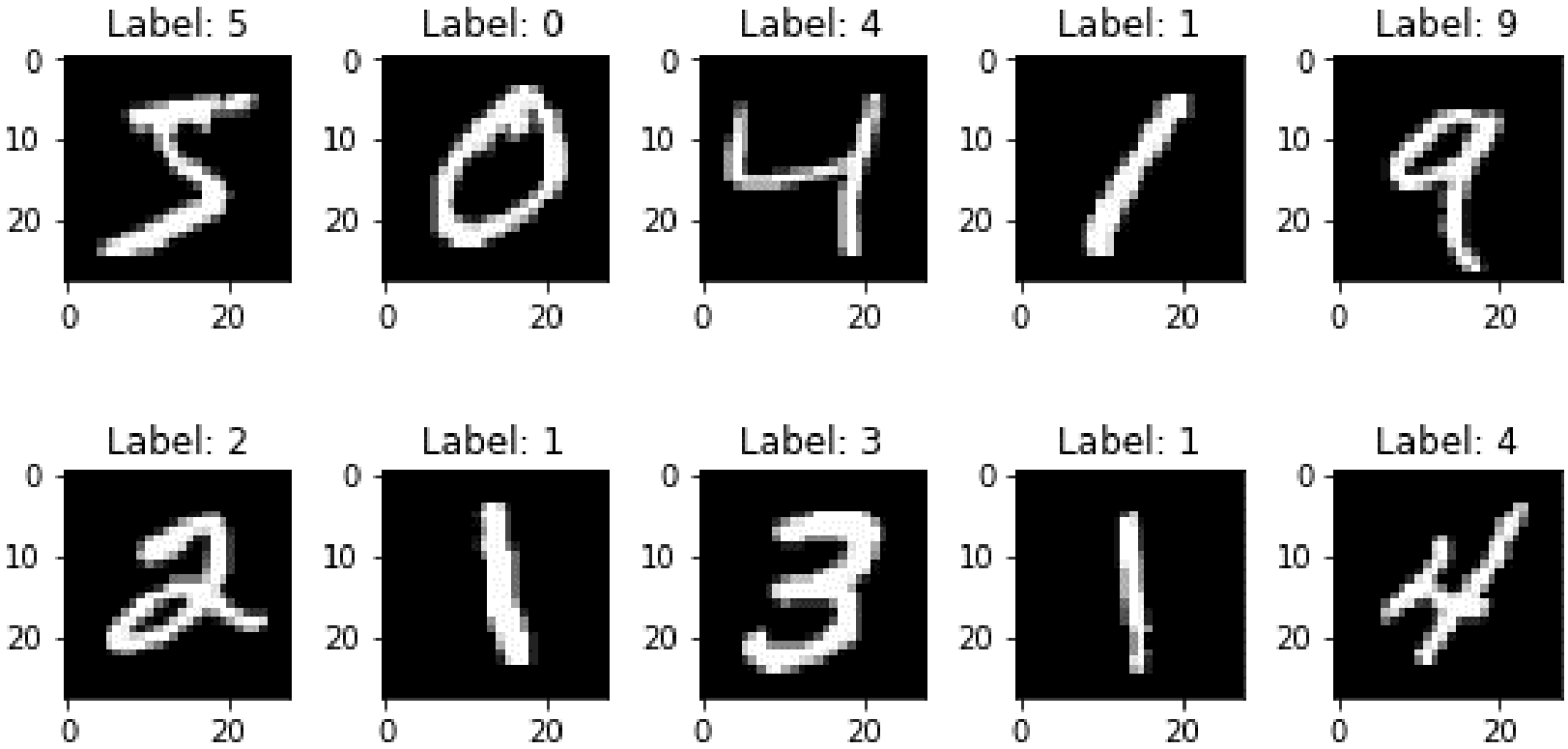
- The problem of automatic recognition of handwritten numbers, also known as digit recognition, is a classic problem in the field of pattern recognition and computer vision. It involves the task of automatically identifying and classifying handwritten digits into their corresponding numerical values.
- The challenge of digit recognition arises due to the variability in handwriting styles and the different ways in which people write the same digit. This variation can be caused by factors such as different writing instruments, writing speed, and writing pressure. Additionally, digits can be written in different sizes, orientations, and positions, further adding to the complexity of the problem.

Introduction to keras: a practical problem

- The problem we are trying to solve here is to classify grayscale images of handwritten digits (28 pixels by 28 pixels), into their 10 categories (0 to 9).
- The dataset we will use is the MNIST dataset, a classic dataset in the machine learning community, which has been around for almost as long as the field itself and has been very intensively studied.
- It's a set of 60,000 training images, plus 10,000 test images, assembled by the National Institute of Standards and Technology (the NIST in MNIST) in the 1980s.

Introduction to keras: a practical problem

- As we have said, in the MNIST dataset each digit is stored in a grayscale image with a size of 28x28 pixels.
- In the following you can see the first 10 digits from the training set:



Introduction to keras: Loading the dataset

- Keras provides seven different datasets, which can be loaded in using Keras directly.
- These include image datasets as well as a house price and a movie review datasets.
- The MNIST dataset comes pre-loaded in Keras, in the form of a set of four Numpy arrays

```
import keras

from keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Introduction to keras

The typical Keras workflows

- Define your training data: input tensor and target tensor
- Define a network of layers(or model) that maps input to our targets.
- Configure the learning process by choosing a loss function, an optimizer, and some metrics to monitor.
- Iterate your training data by calling the fit() method of your model.

Introduction to keras: Layers

- This is the building block of neural networks which are stacked or combined together to form a neural network model.
- It is a data-preprocessing module that takes one or more input tensors and outputs one or more tensors.
- These layers together contain the network's knowledge.
- Different layers are made for different tensor formats and data processing.

Creating a model with the sequential API

- The easiest way of creating a model in Keras is by using the sequential API, which lets you stack one layer after the other.
- The problem with the sequential API is that it doesn't allow models to have multiple inputs or outputs, which are needed for some problems.
- Nevertheless, the sequential API is a perfect choice for most problems.

```
from keras import models
from keras import layers

network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

Introduction to keras

Let's go through this code line by line:

- The first line creates a Sequential model.
- This is the simplest kind of Keras model, for neural networks that are just composed of a single stack of layers, connected sequentially.
- This is called the **sequential** API.

```
from keras import models
from keras import layers

network = models.Sequential()
```

Introduction to keras

Next, we build the first layer and add it to the model.

- It is **Dense** hidden layer with 512 neurons.
- It will use the ReLU activation function.
- Each Dense layer manages its own weight matrix, containing all the connection weights between the neurons and their inputs.
- It also manages a vector of bias terms (one per neuron).
- When it receives some input data, it computes

$$\phi\left(Z^{[1]} = W^{[1]} \cdot X + B^{[1]}\right), \quad \phi(z) = ReLU(z)$$

```
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
```

Introduction to keras

Finally, we add a Dense output layer with 10 neurons (one per class).

- Using a 10-way "softmax" layer means that it will return an array of 10 probability scores (summing to 1).
- Each score will be the probability that the current digit image belongs to one of our 10 digit classes.

```
network.add(layers.Dense(10, activation='softmax'))
```

Introduction to keras

- The model's summary() method displays all the model's layers, including each layer's name (which is automatically generated unless you set it when creating the layer), its output shape (None means the batch size can be anything), and its number of parameters.

```
network.summary()
```

Introduction to keras

- The summary ends with the total number of parameters, including trainable and non-trainable parameters.
- Here we only have trainable parameters.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 512)               401920
_________________________________________________________________
dense_3 (Dense)              (None, 10)                5130
=================================================================
Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
_________________________________________________________________
```

Compile a Model

- Before we can start training our model we need to configure the learning process.
- For this, we need to specify an optimizer, a loss function and optionally some metrics like accuracy.
- The **loss function** is a measure on how good our model is at achieving the given objective.
- An **optimizer** is used to minimize the loss(objective) function by updating the weights using the gradients.

Loss Function

- Choosing the right Loss Function for the problem is very important, the neural network can take any shortcut to minimize the loss.
- So, if the objective doesn't fully correlate with success for the task at hand, your network will end up doing things you may not have wanted.

Loss Function

For common problems like Classification, Regression and Sequence prediction, they are simple guidelines to choose a loss function.

For:

- Two- Class classification you can choose binary cross-entropy
- Multi-Class Classification you can choose Categorical Cross-entropy.
- Regression Problem you can choose Mean-Squared Error

Activation Function

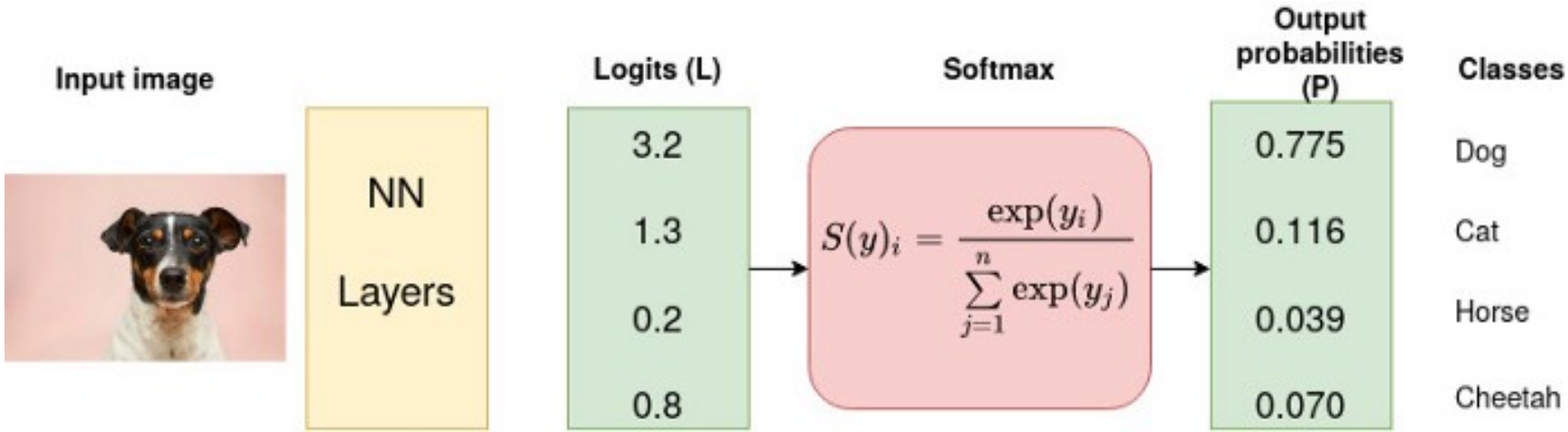
Softmax Activation Function

- Softmax is an activation function that scales numbers/logits into probabilities.
- The output of a Softmax is a vector (say v) with probabilities of each possible outcome.
- The probabilities in vector v sums to one for all possible outcomes or classes
- It is often used as the last activation function of a neural network to normalize the output of a network to a probability distribution over predicted output classes
- You can think of softmax function as a sort of generalization to multiple dimension of the logistic function

Activation Function

Softmax Activation Function. Mathematically, softmax is defined as

$$S(y)_i = \frac{\exp y_i}{\sum_{j=1}^{n} \exp y_j} \tag{13}$$



Loss Function

Categorical Cross-Entropy

- Remember the logistic Loss Function:

$$L(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}) = -\sum_{j=1}^{2} y_j \log \hat{y}_j$$

- Generalizing this result to the n-class classification problem we have

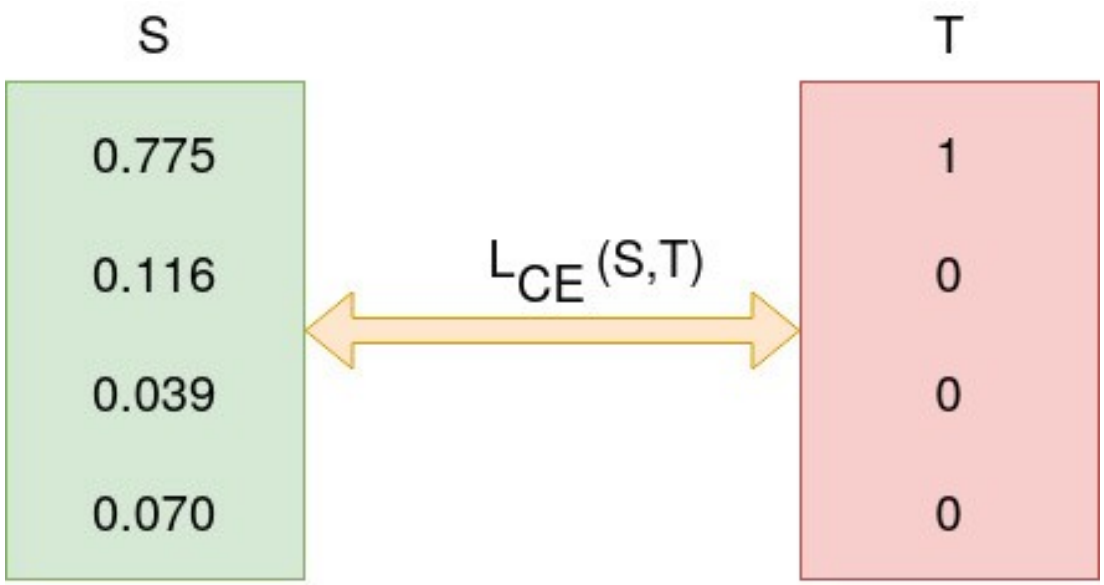
$$L = -\sum_{j=1}^{N_C} y_j \log \hat{y}_j \tag{14}$$

This last equation define the so called *cross-entropy*

Loss Function

Categorical Cross-Entropy

- In the previous example, Softmax converts logits into probabilities.
- The purpose of the Cross-Entropy is to take the output probabilities (P) and measure the distance from the truth values



Loss Function

Categorical Cross-Entropy

- The categorical cross-entropy is computed as follows

$$
\begin{aligned}
L_{CE} &= -\sum_{i=1} T_i \log(S_i) \\
&= -\left[1 \log_2(0.775) + 0 \log_2(0.126) + 0 \log_2(0.039) + 0 \log_2(0.070)\right] \\
&= -\log_2(0.775) \\
&= 0.3677
\end{aligned}
$$

Optimizer

RMSProp

- Gradient descent is an optimization algorithm that follows the negative gradient of an objective function in order to locate the minimum of the function.
- A limitation of gradient descent is that it uses the same step size (learning rate) for each input variable.
- AdaGrad is an extension of the gradient descent optimization algorithm that allows the step size in each dimension used by the optimization algorithm to be automatically adapted based on the gradients seen for the variable (partial derivatives) over the course of the search.
- A limitation of AdaGrad is that it can result in a very small step size for each parameter by the end of the search that can slow the progress of the search down too much and may mean not locating the optima.
- Root Mean Squared Propagation, or RMSProp, is an extension of gradient descent and the AdaGrad version of gradient descent that uses a decaying average of partial gradients in the adaptation of the step size for each parameter.
- The use of a decaying moving average allows the algorithm to forget early gradients and focus on the most recently observed partial gradients seen during the progress of the search, overcoming the limitation of AdaGrad.

Compile the model

So, to make our network ready for training, we need to pick three things, as part of "compilation" step:

- A loss function: the is how the network will be able to measure how good a job it is doing on its training data, and thus how it will be able to steer itself in the right direction.
- An optimizer: this is the mechanism through which the network will update itself based on the data it sees and its loss function.
- Metrics to monitor during training and testing. Here we will only care about accuracy (the fraction of the images that were correctly classified).

```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

Introduction to keras: Training

- Before training, we will preprocess our data by reshaping it into the shape that the network expects, and scaling it so that all values are in the ‘[0, 1]‘ interval.
- Previously, our training images for instance were stored in an array of shape ‘(60000, 28, 28)‘ of type ‘uint8‘ with values in the ‘[0, 255]‘ interval.
- We transform it into a ‘float32‘ array of shape ‘(60000, 28 * 28)‘ with values between 0 and 1.

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

Introduction to keras: Training

We also need to categorically encode the labels:

```
from keras.utils.np_utils import to_categorical

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

Introduction to keras: Training

What is an epoc?

- An epoch is a term used in machine learning and indicates the number of passes of the entire training dataset the machine learning algorithm has completed. Datasets are usually grouped into batches (especially when the amount of data is very large). Some people use the term iteration loosely and refer to putting one batch through the model as an iteration.
- If the batch size is the whole training dataset then the number of epochs is the number of iterations. For practical reasons, this is usually not the case. Many models are created with more than one epoch. The general relation where dataset size is d, number of epochs is e, number of iterations is i, and batch size is b would be $d \cdot e = i \cdot b$.
- Determining how many epochs a model should run to train is based on many parameters related to both the data itself and the goal of the model, and while there have been efforts to turn this process into an algorithm, often a deep understanding of the data itself is indispensable.

Introduction to keras: Training

We are now ready to train our network, which in Keras is done via a call to the 'fit' method of the network: we "fit" the model to its training data.

```
history = network.fit(train_images, train_labels, epochs=5, batch_size=128)
```

Visualizing the training process

- We can visualize our training and testing accuracy and loss for each epoch so we can get intuition about the performance of our model.
- The accuracy and loss over epochs are saved in the history variable we got whilst training and we will use Matplotlib to visualize this data.

```
# list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['mae'])
plt.plot(history.history['val_mae'])
plt.title('model mae')
plt.ylabel('mae')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

Other Neural Network Architectures

Subsection 1

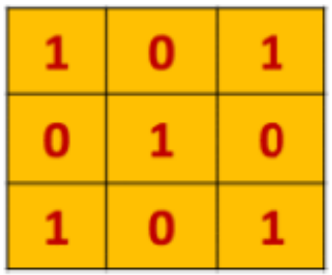
Convolutional Neural Network

Convolutional Neural Network

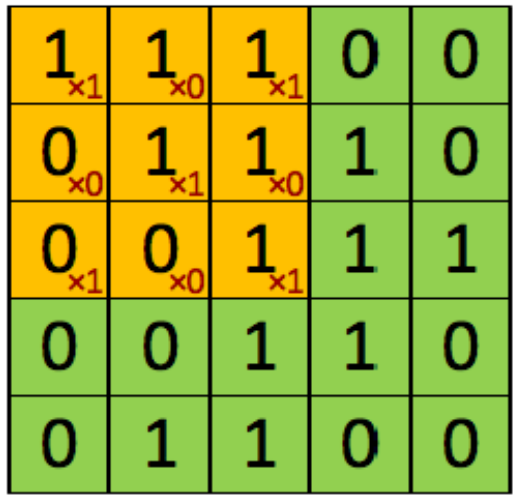
- A convolutional neural network, or CNN for short, is a type of artificial neural network that is commonly used for image recognition and classification tasks.
- It is inspired by the organization of the visual cortex in animals and is designed to process visual data in a way that is similar to how the human brain processes visual information.
- The main idea behind a CNN is to use a series of **convolutional layers** to automatically learn features from the input image, such as edges, lines, and shapes.
- These features are then combined and fed into a series of fully connected layers that perform the final classification task.

Convolutional Layer

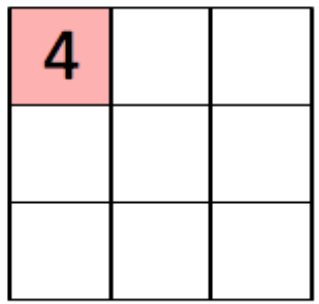
- A **convolutional layer** is a key building block in convolutional neural networks (CNNs) used for image recognition and computer vision tasks.
- It performs a mathematical operation called convolution on the input image to extract features that are useful for the given task.



Filter / Kernel



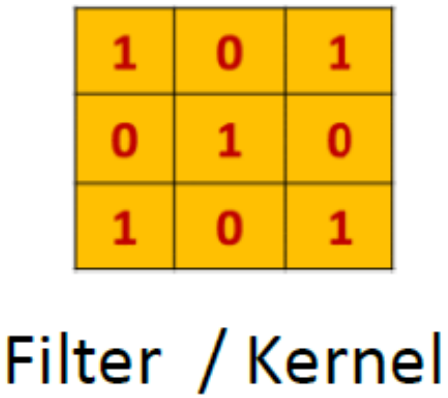
Image

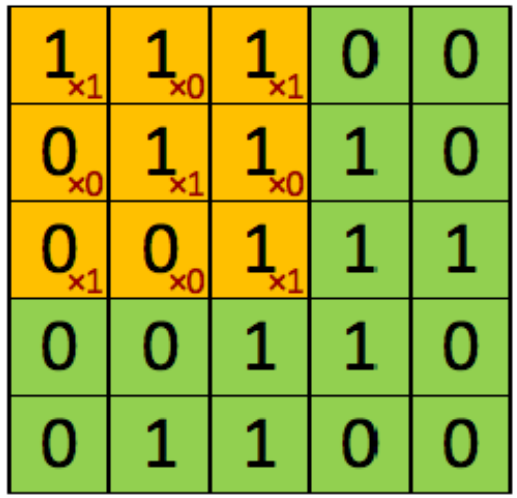


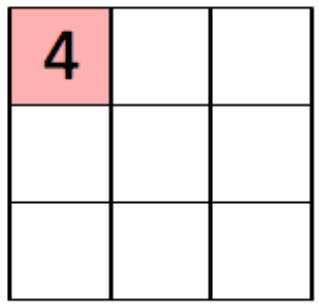
Convolved Feature

What is Convolution

- The convolution operation involves sliding a small matrix of numbers called a **kernel** or **filter** over the input image.
- The kernel is typically much smaller than the input image and contains learnable weights that are optimized during the training process.







What is a Convolution

Convolutional layers typically have multiple kernels that are applied to the input image to extract different features. These features may include edges, textures, and patterns that are relevant to the given image recognition task.

Original

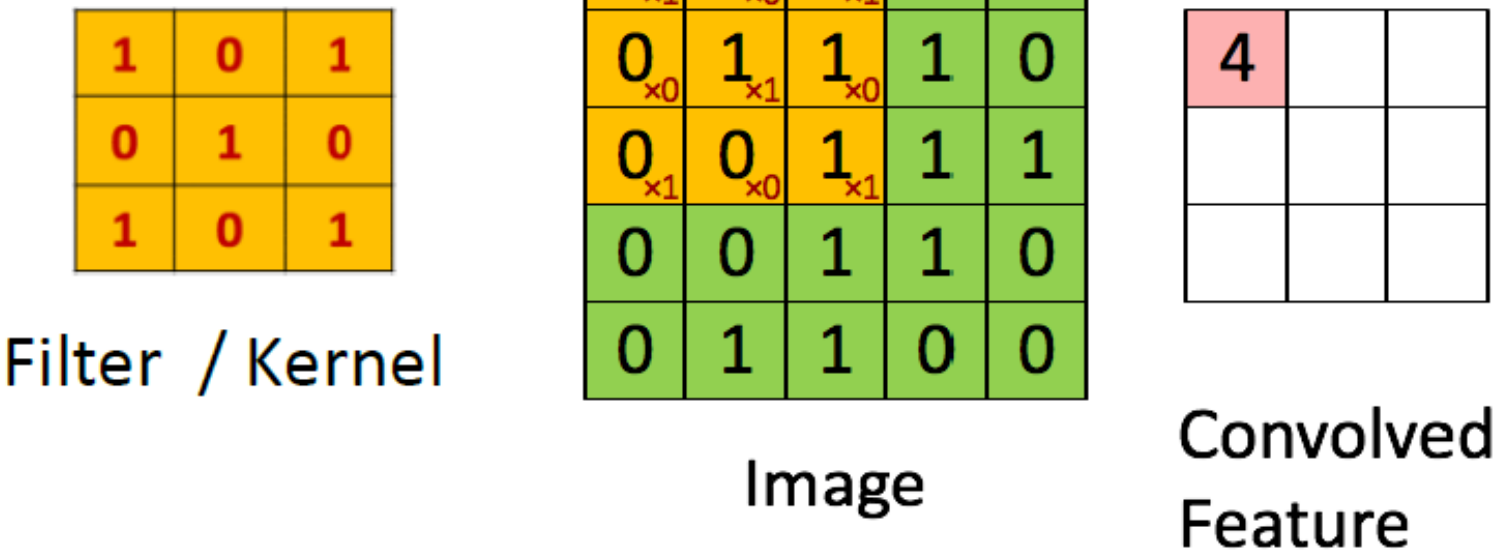
Sharpen

Edge Detect

"Strong" Edge Detect

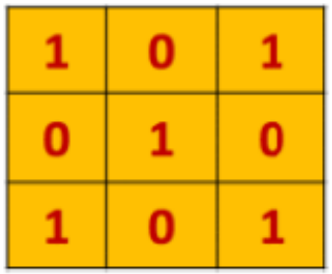
What is Convolution

- An easy way to understand this is if you were a detective and you are came across a large image or a picture in dark, how will you identify the image?
- You will use you flashlight and scan across the entire image. This is exactly what we do in convolutional layer.
- Kernel K, which is a feature detector is equivalent of the flashlight on image I, and we are trying to detect feature to help us identify or classify the image.

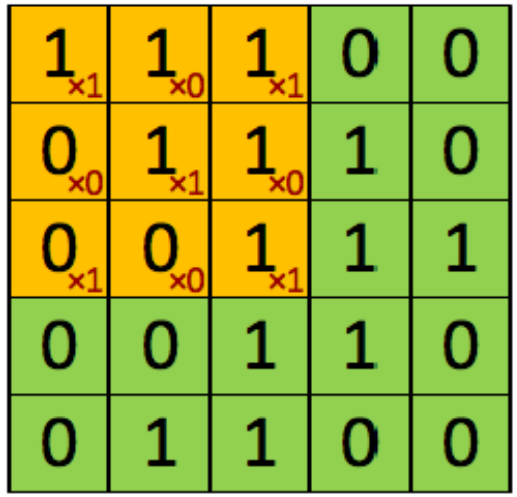


What is Convolution

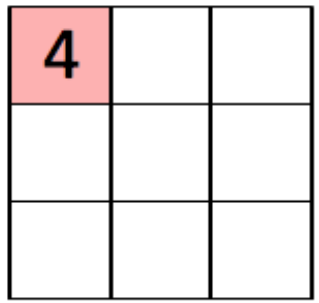
- As the kernel slides over the input image, it computes a dot product between its weights and the pixel values of the corresponding region of the image.
- The output of this dot product is a single number that is used to create a new output image or feature map.
- Just like the weights in a fully connected layer, the kernel weights are learned during training, and adjusted after each training iteration through backpropagation.



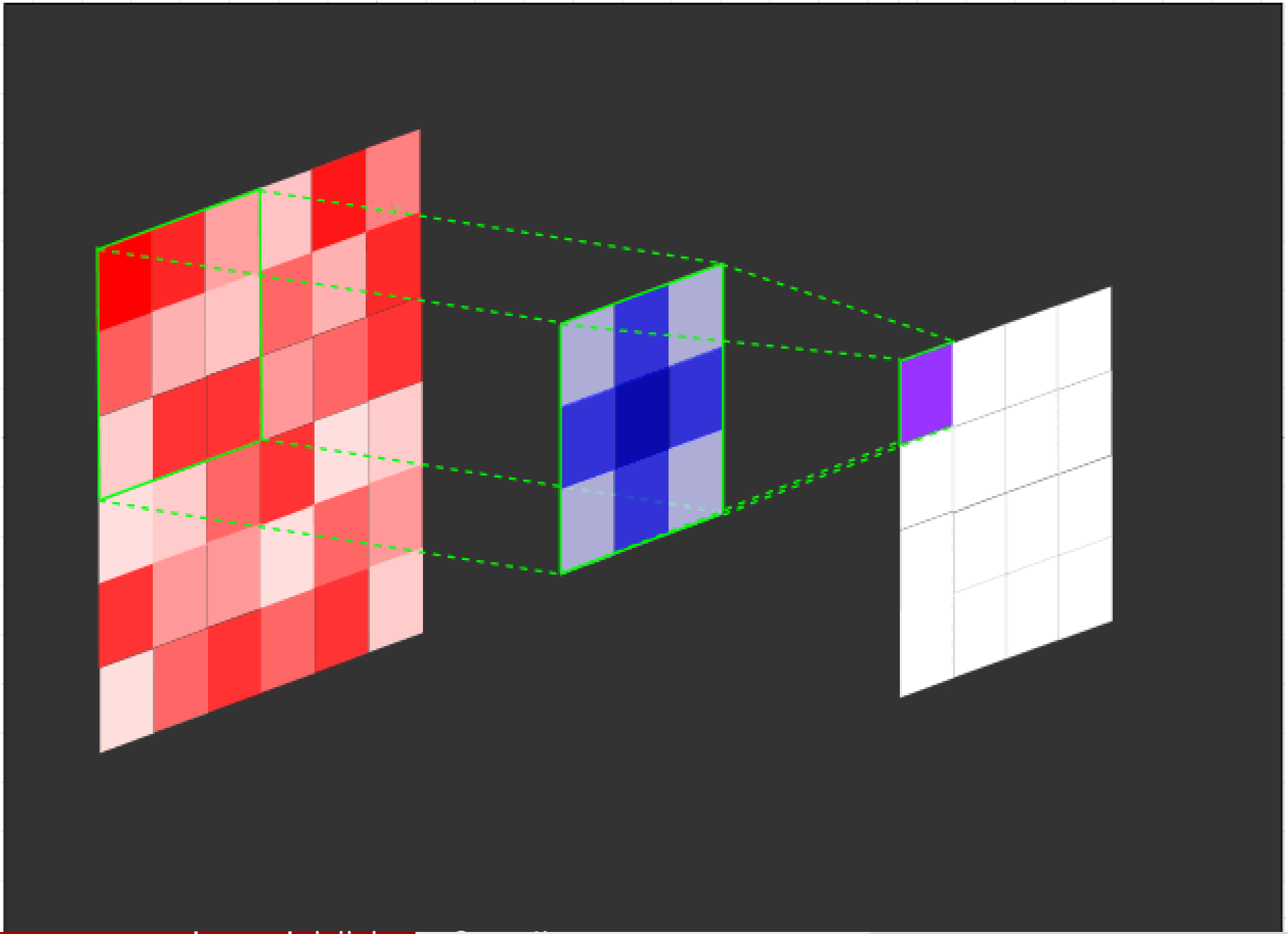
Filter / Kernel

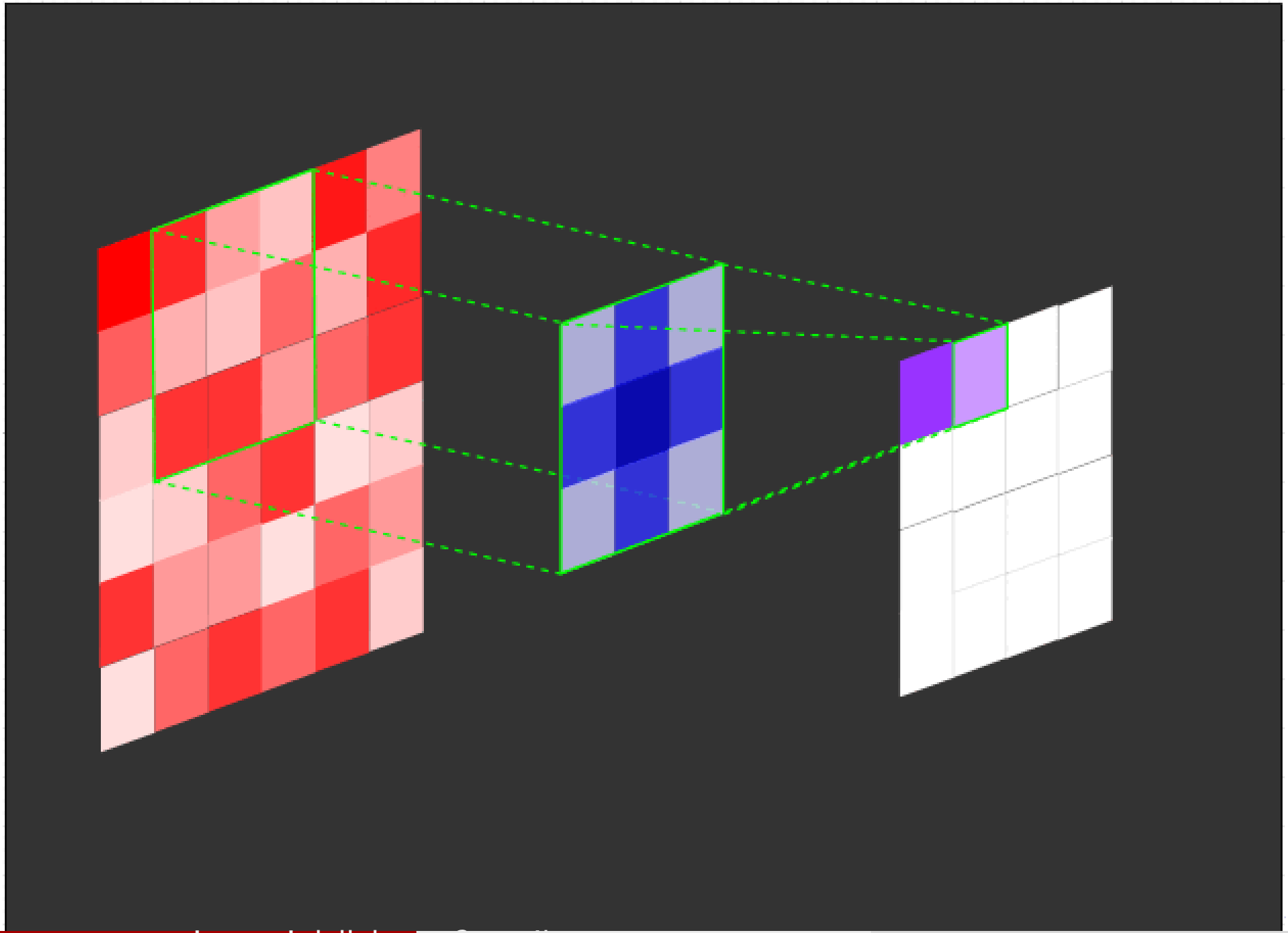


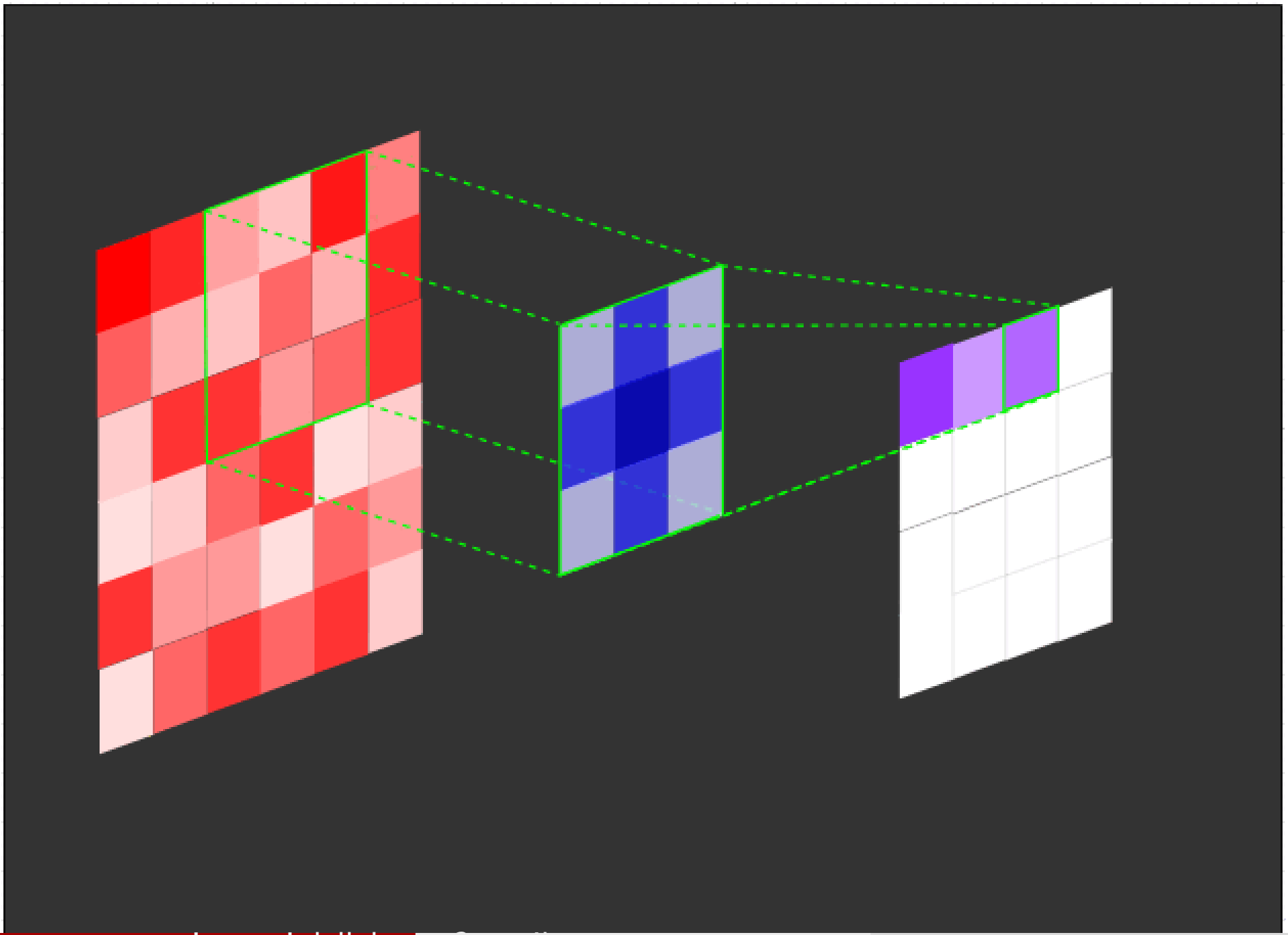
Image

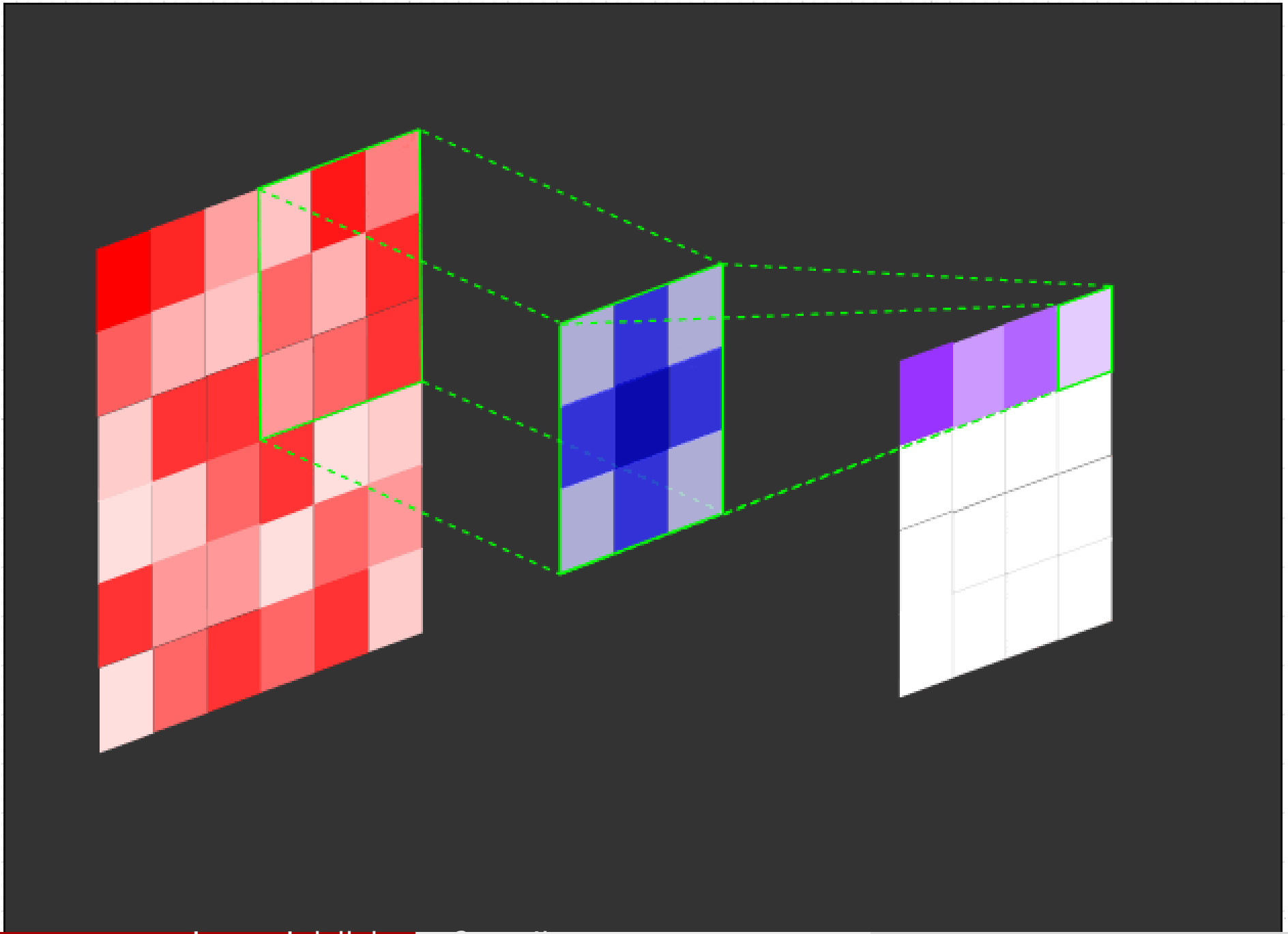


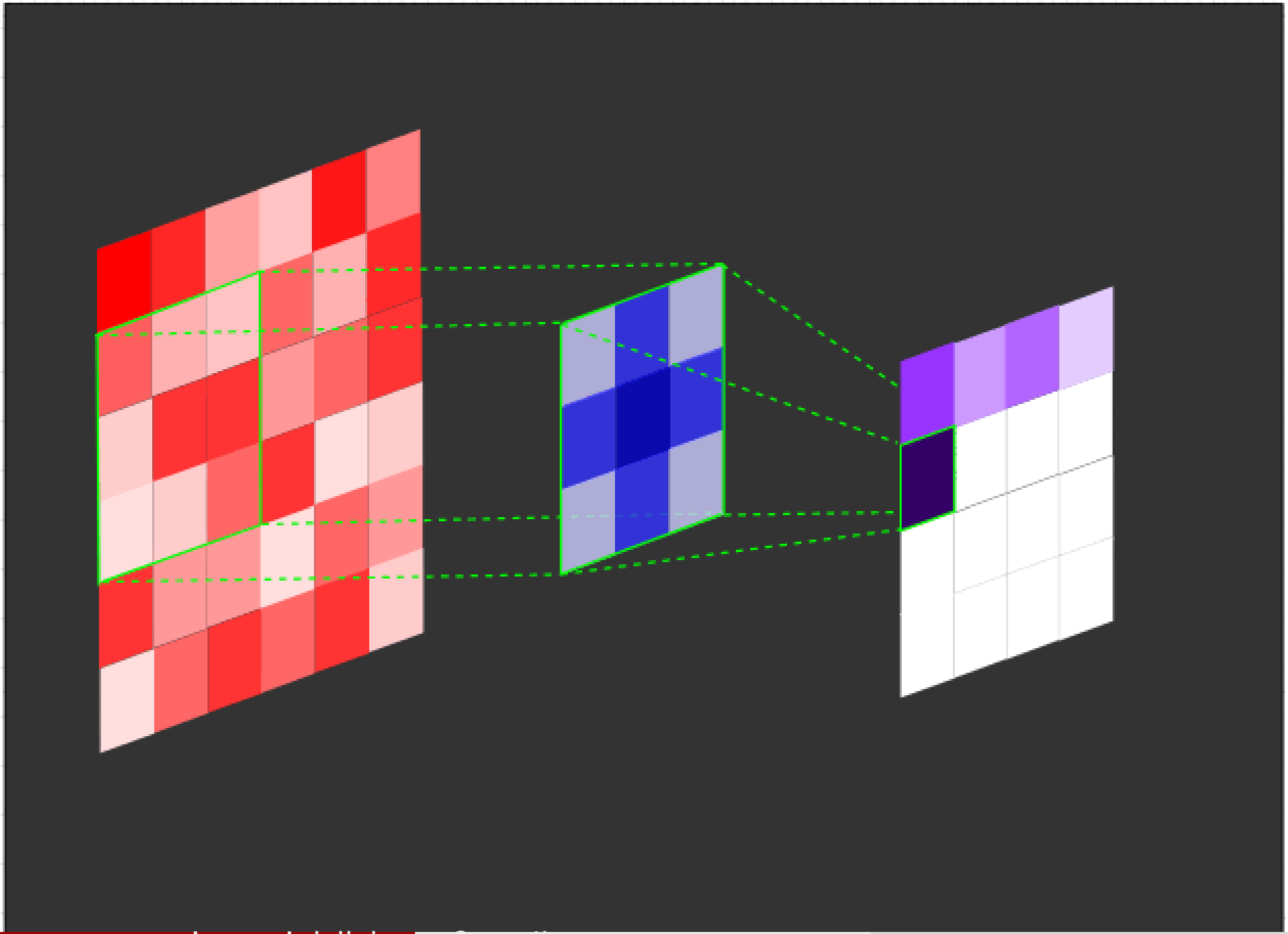
Convolved Feature

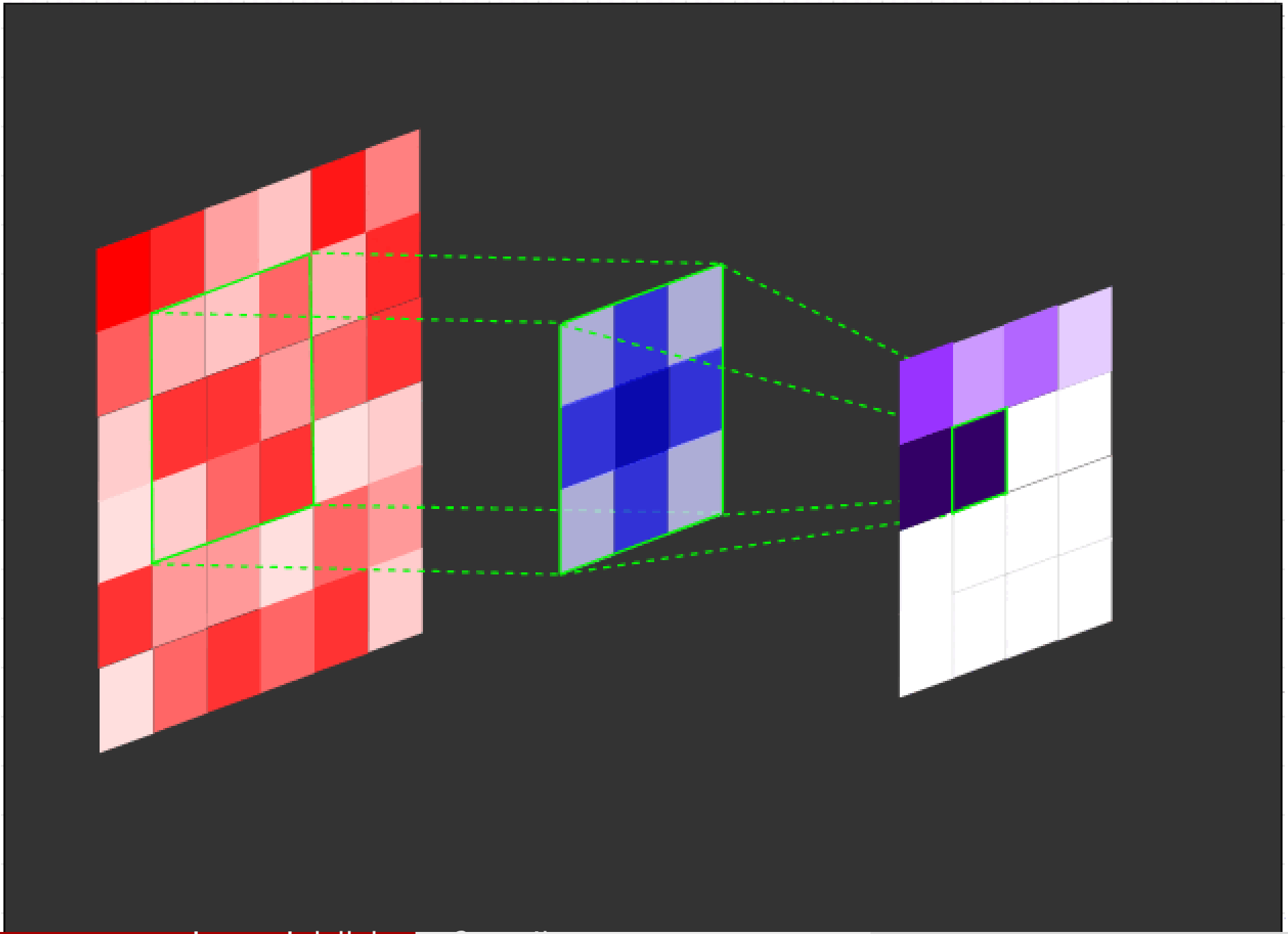


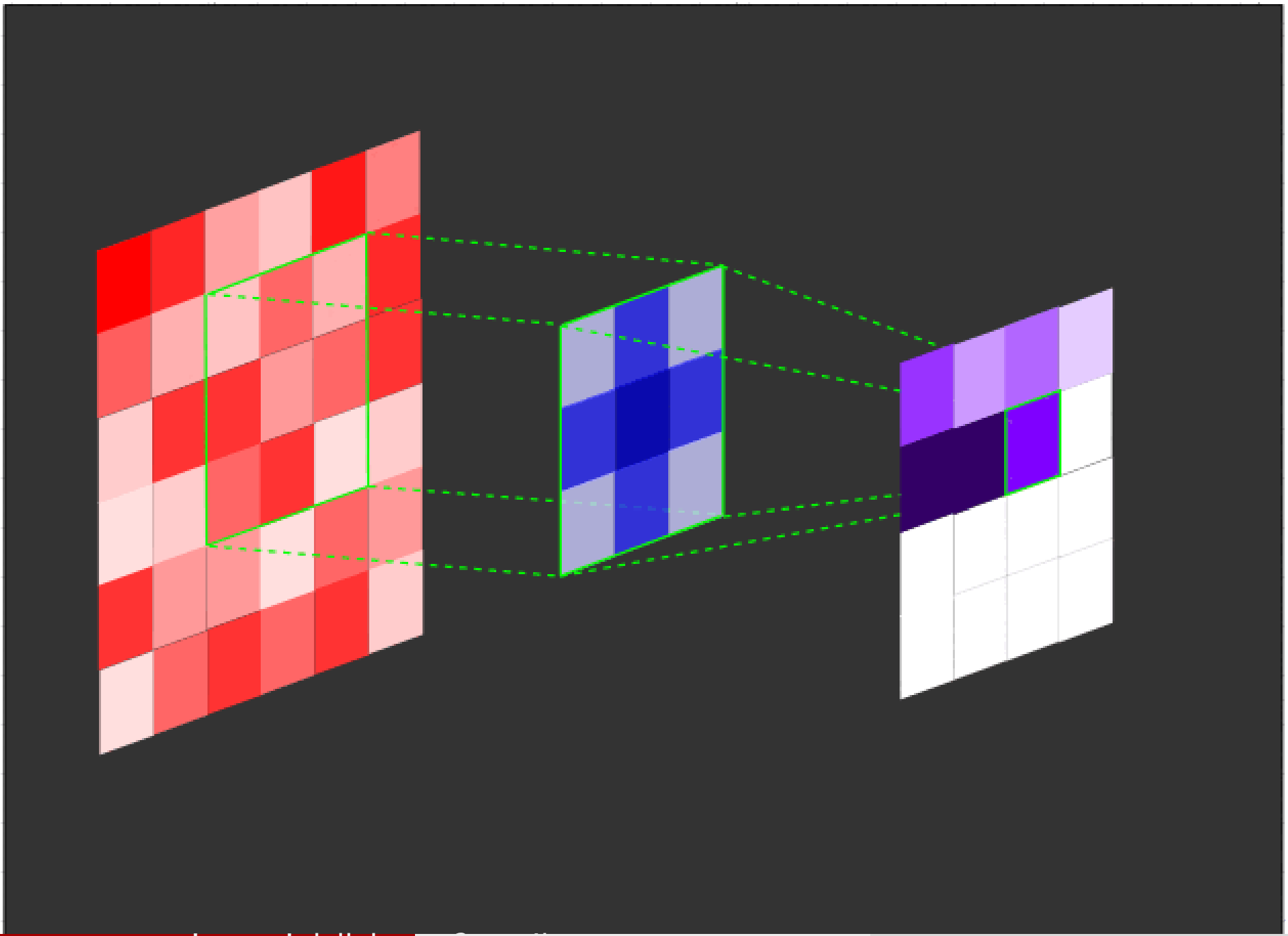


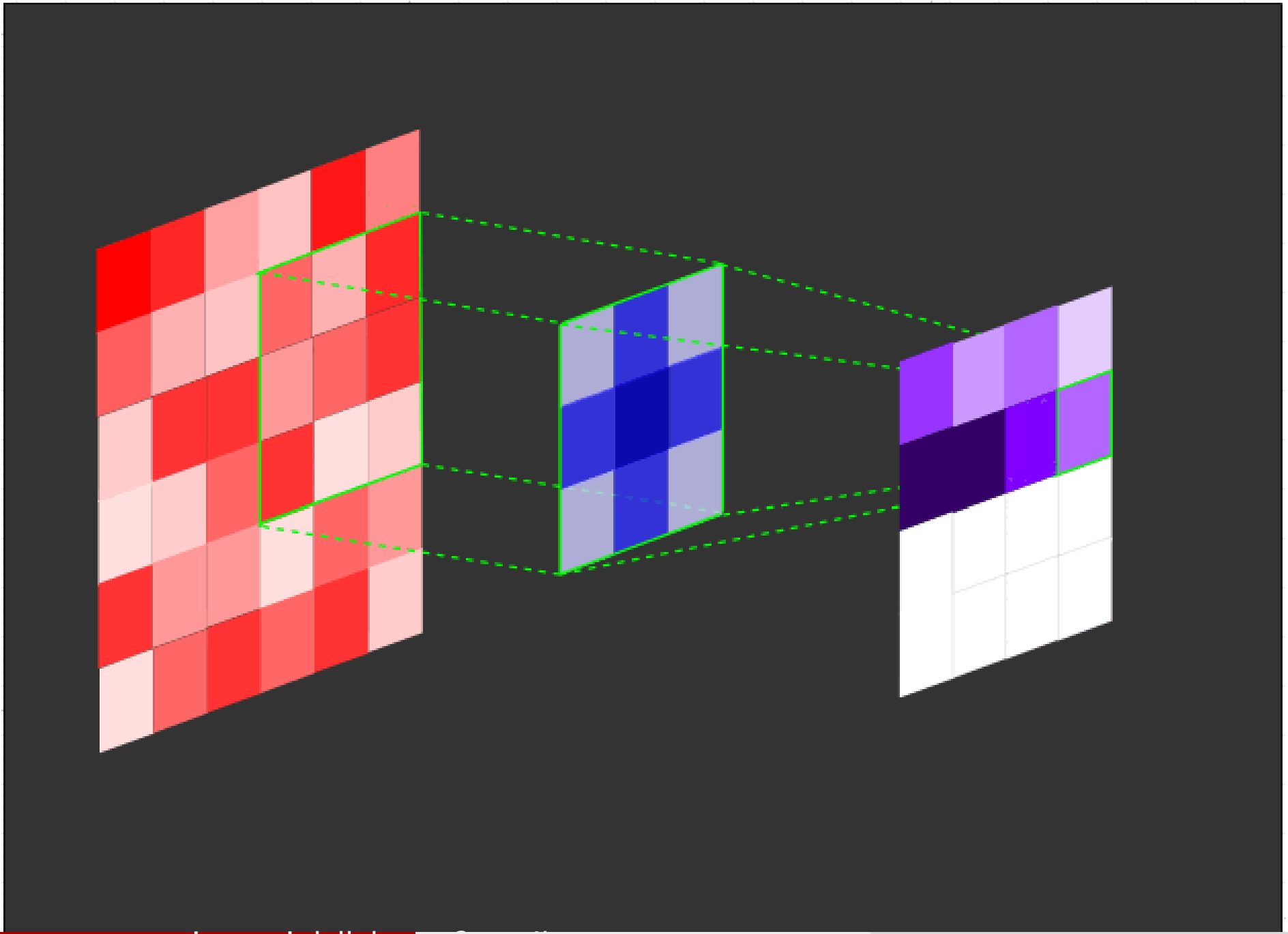


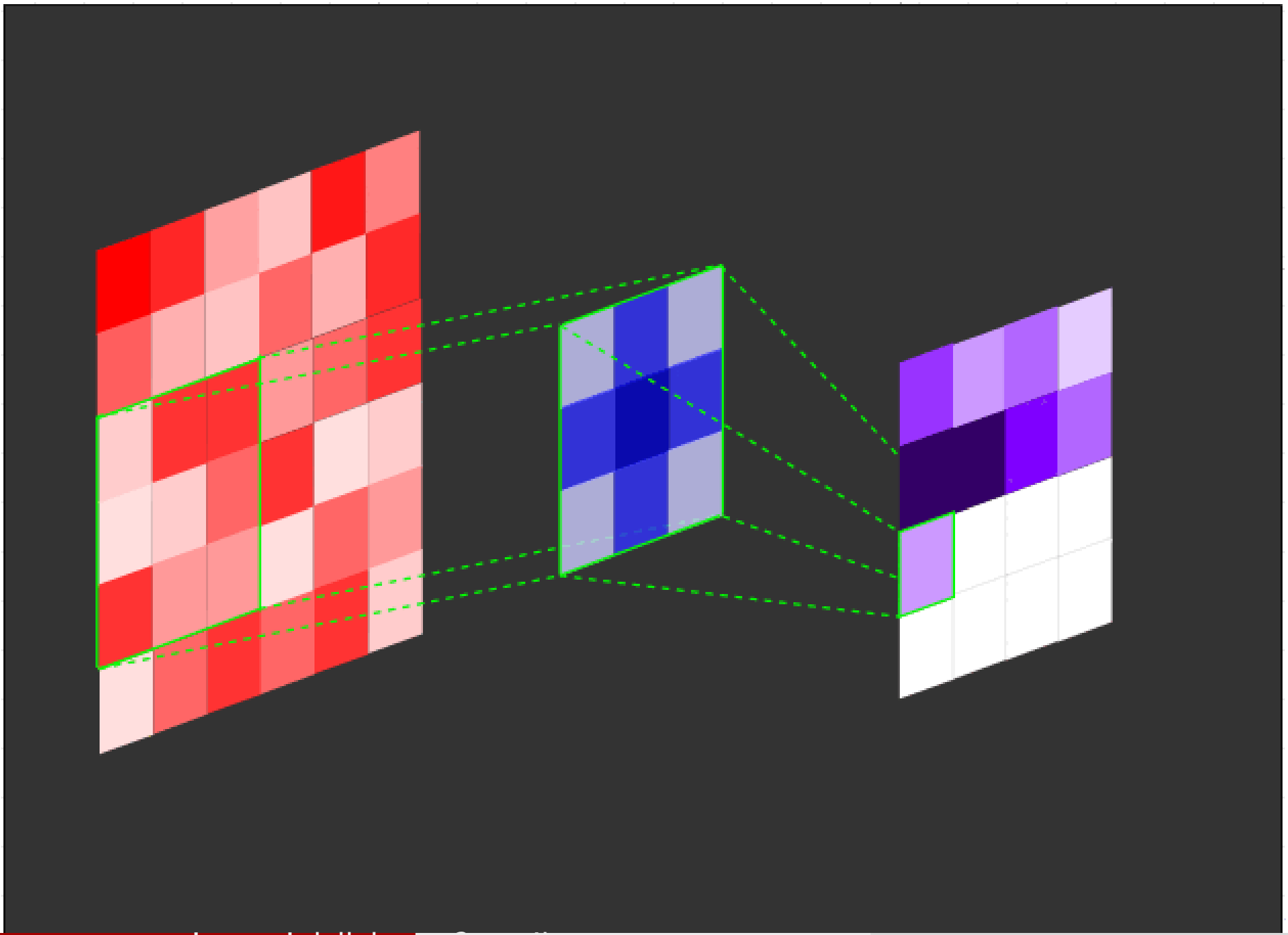


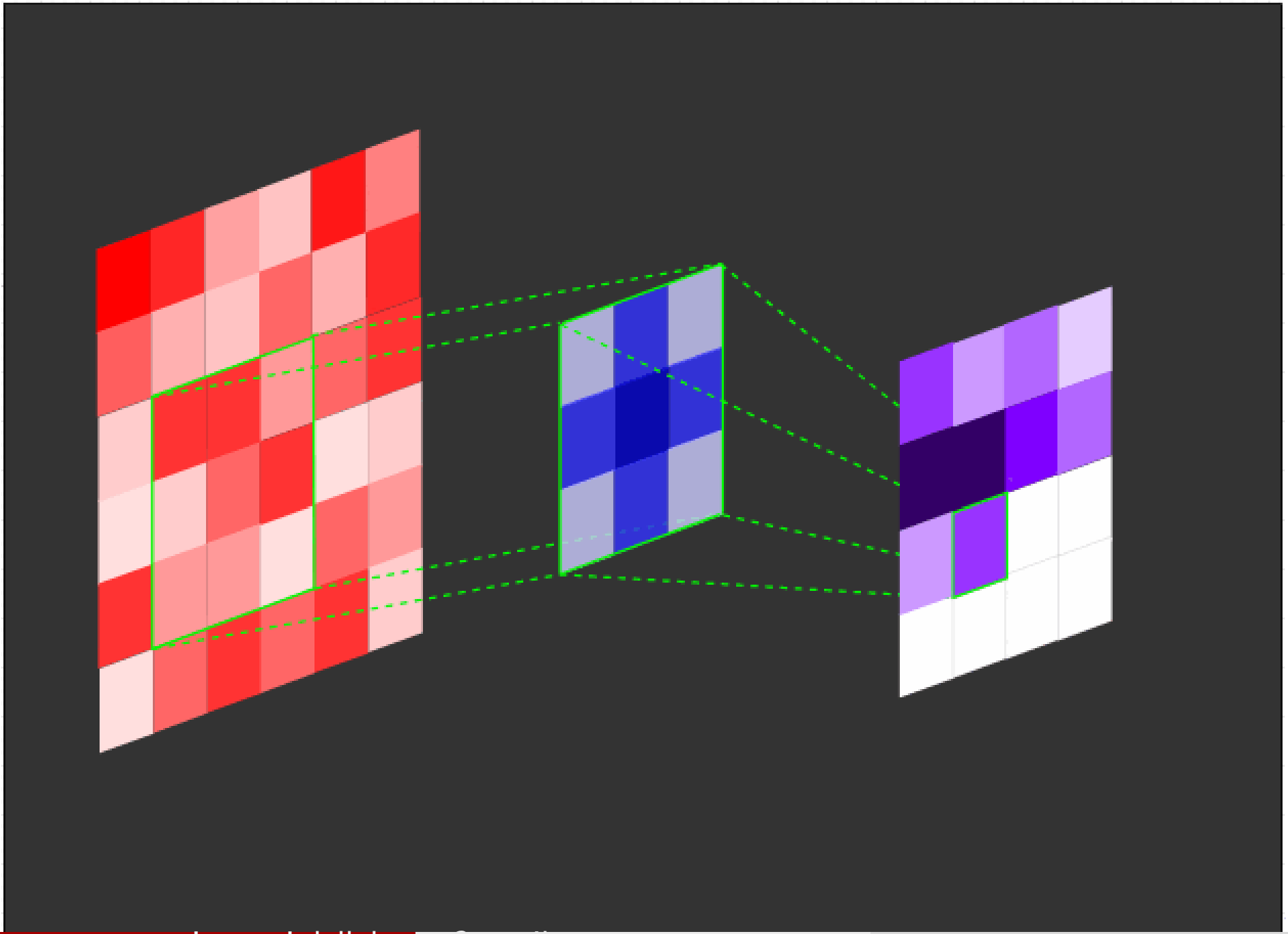


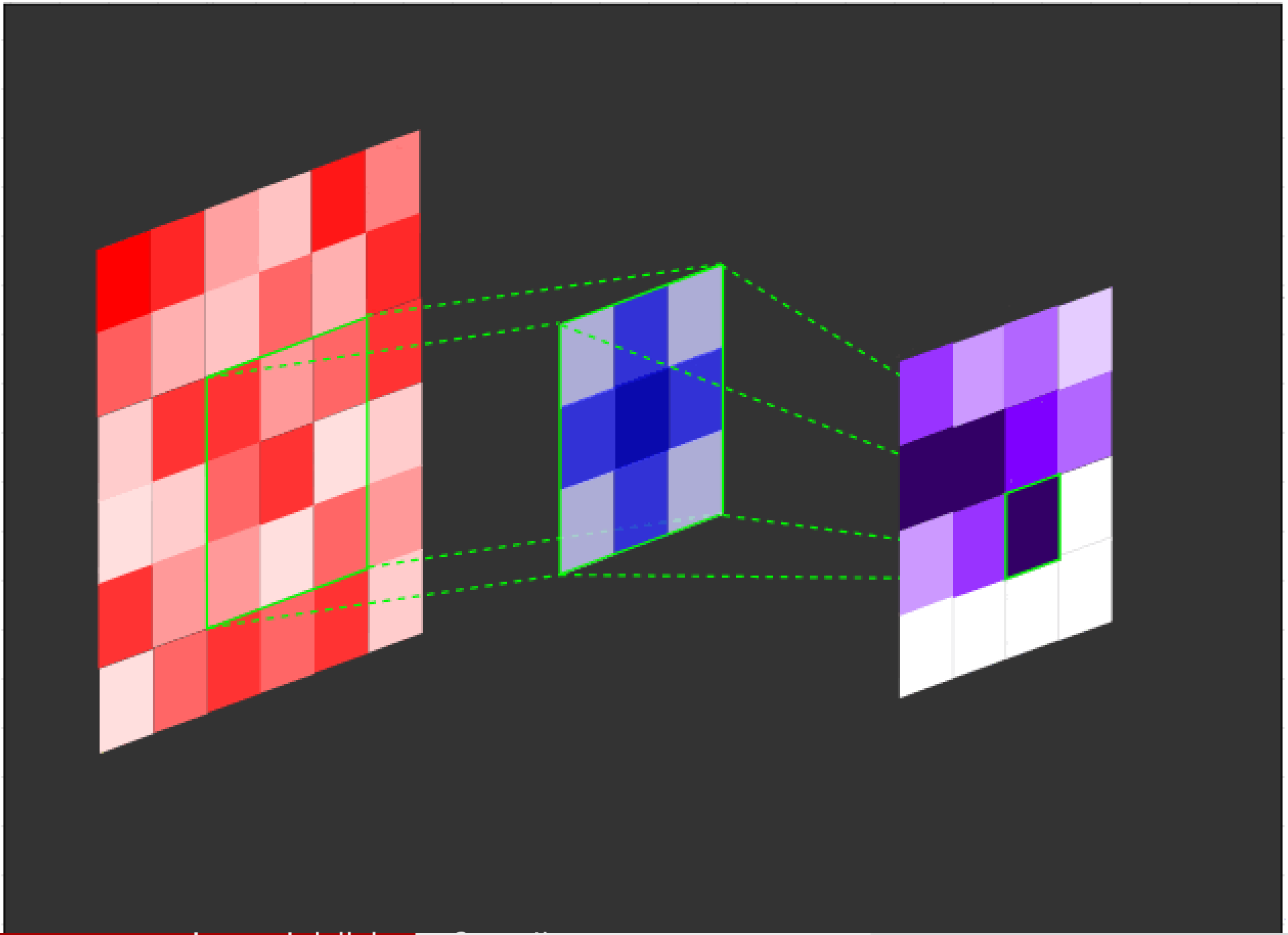


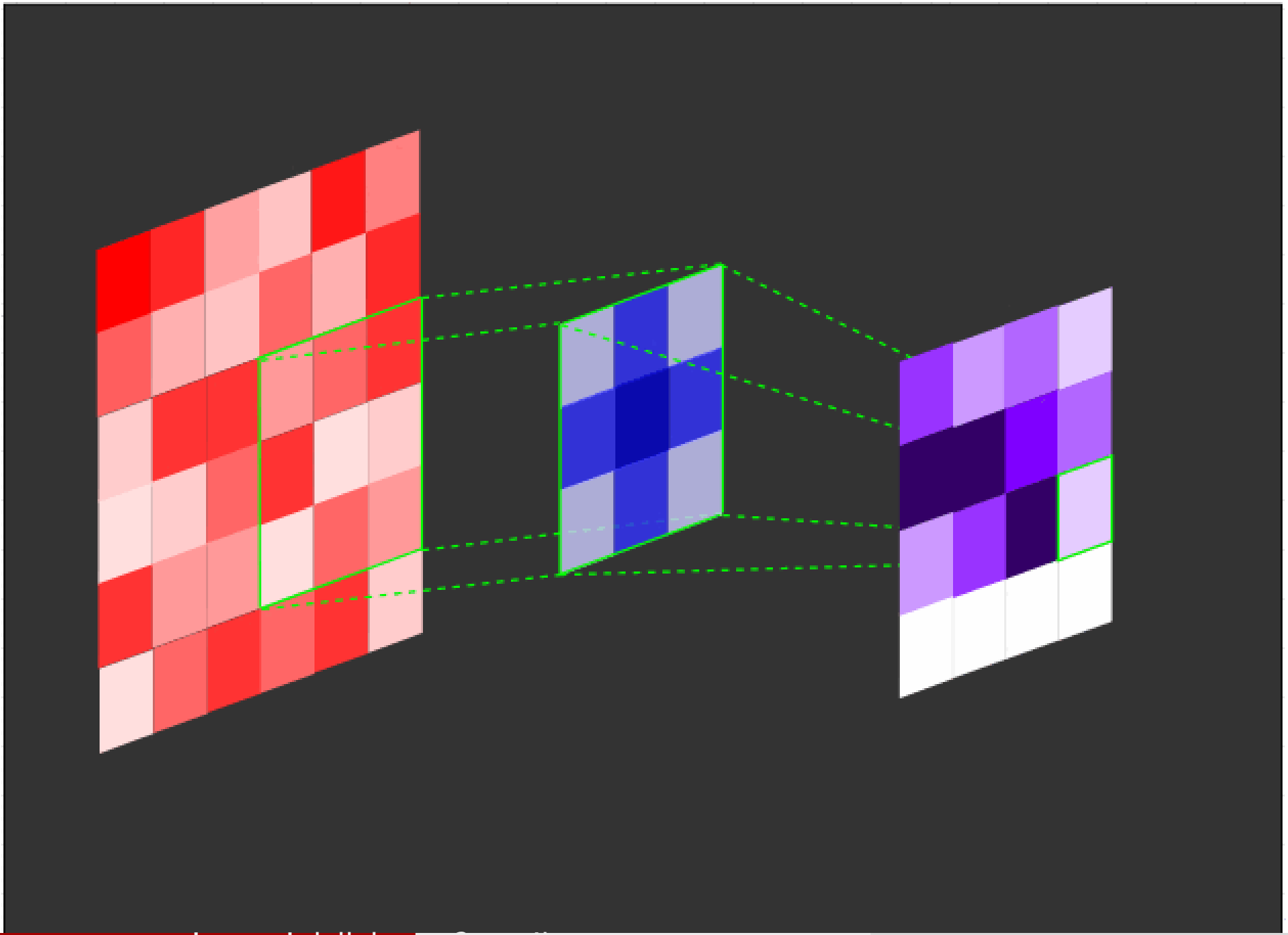


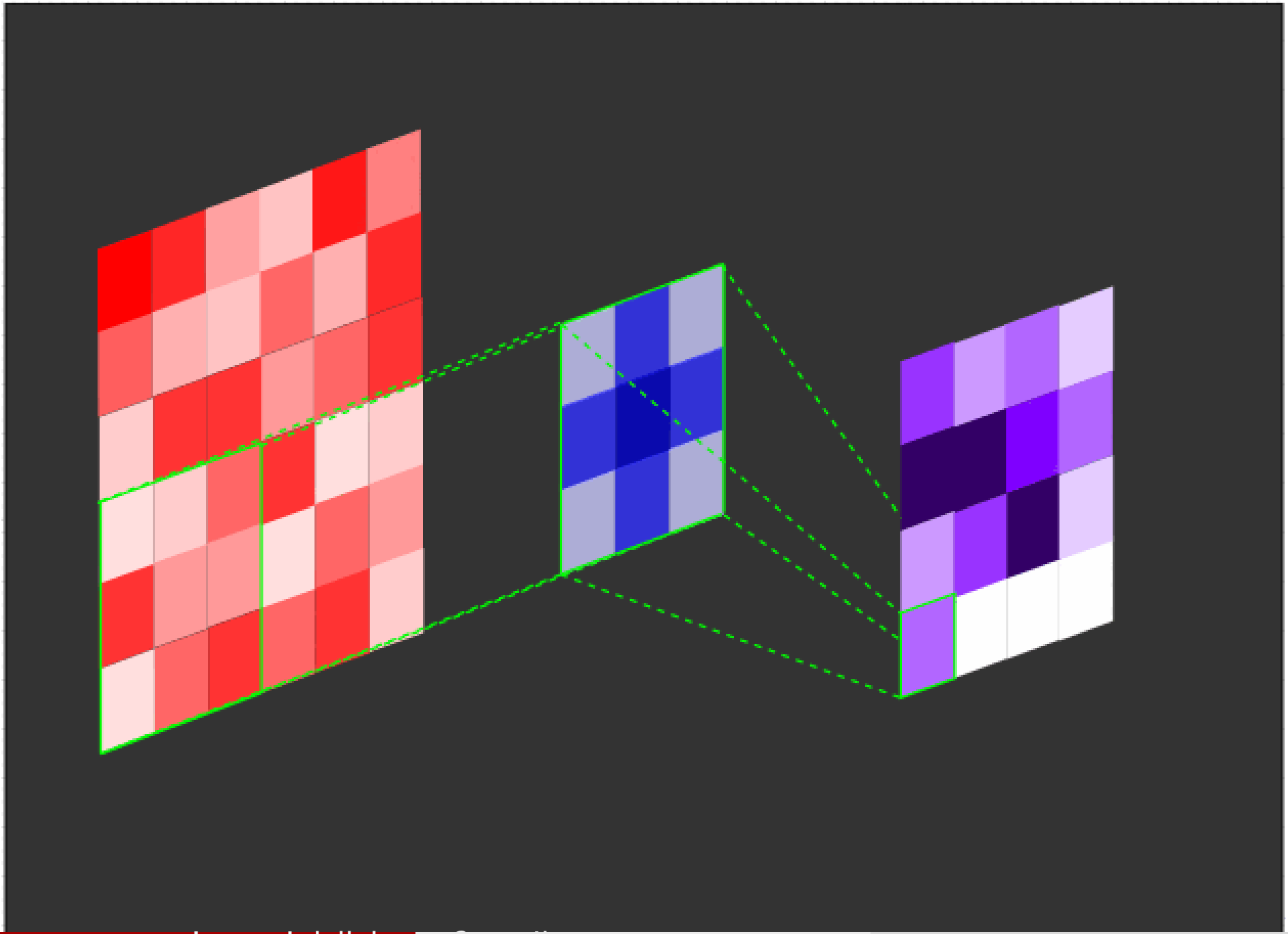


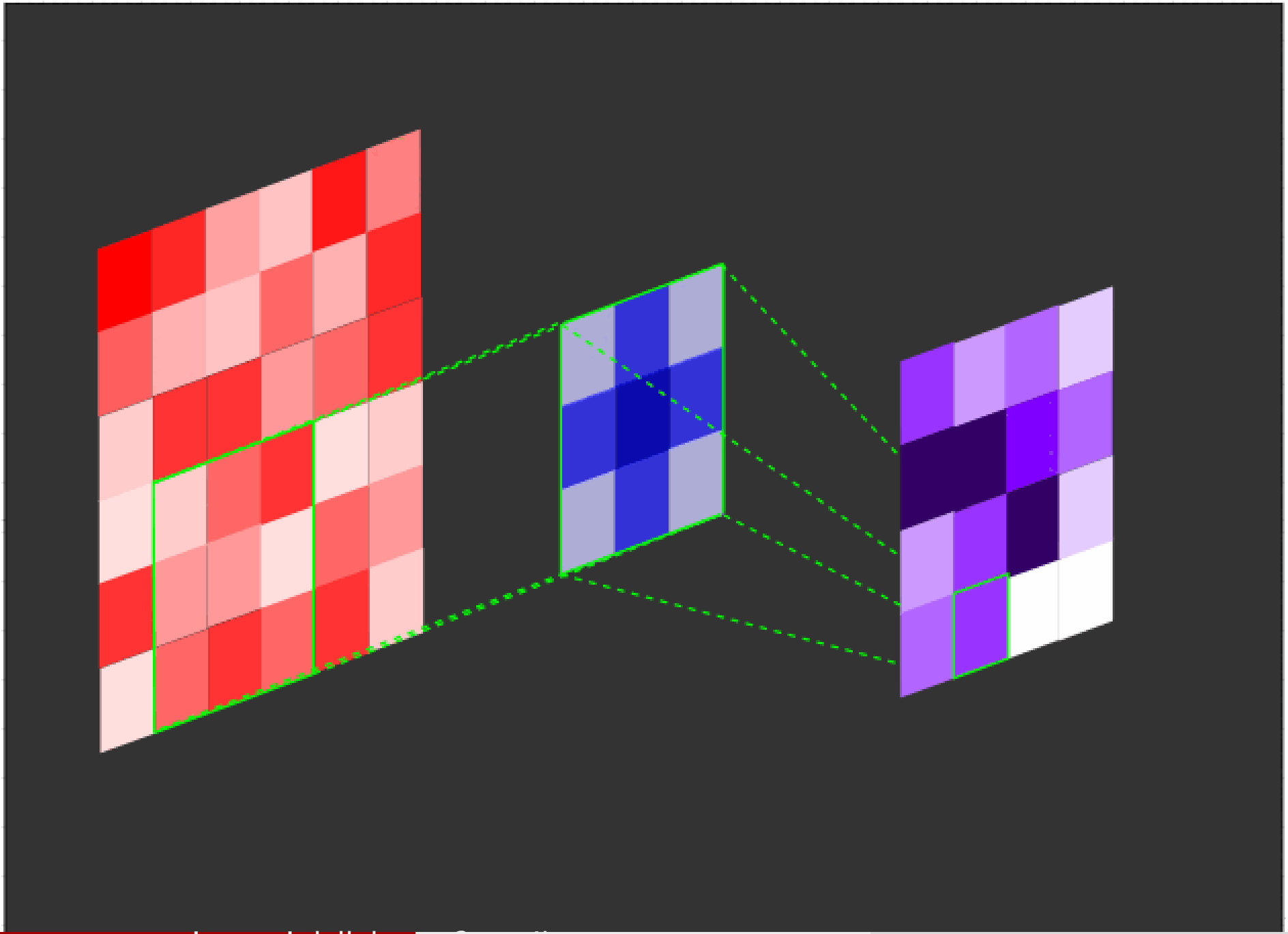


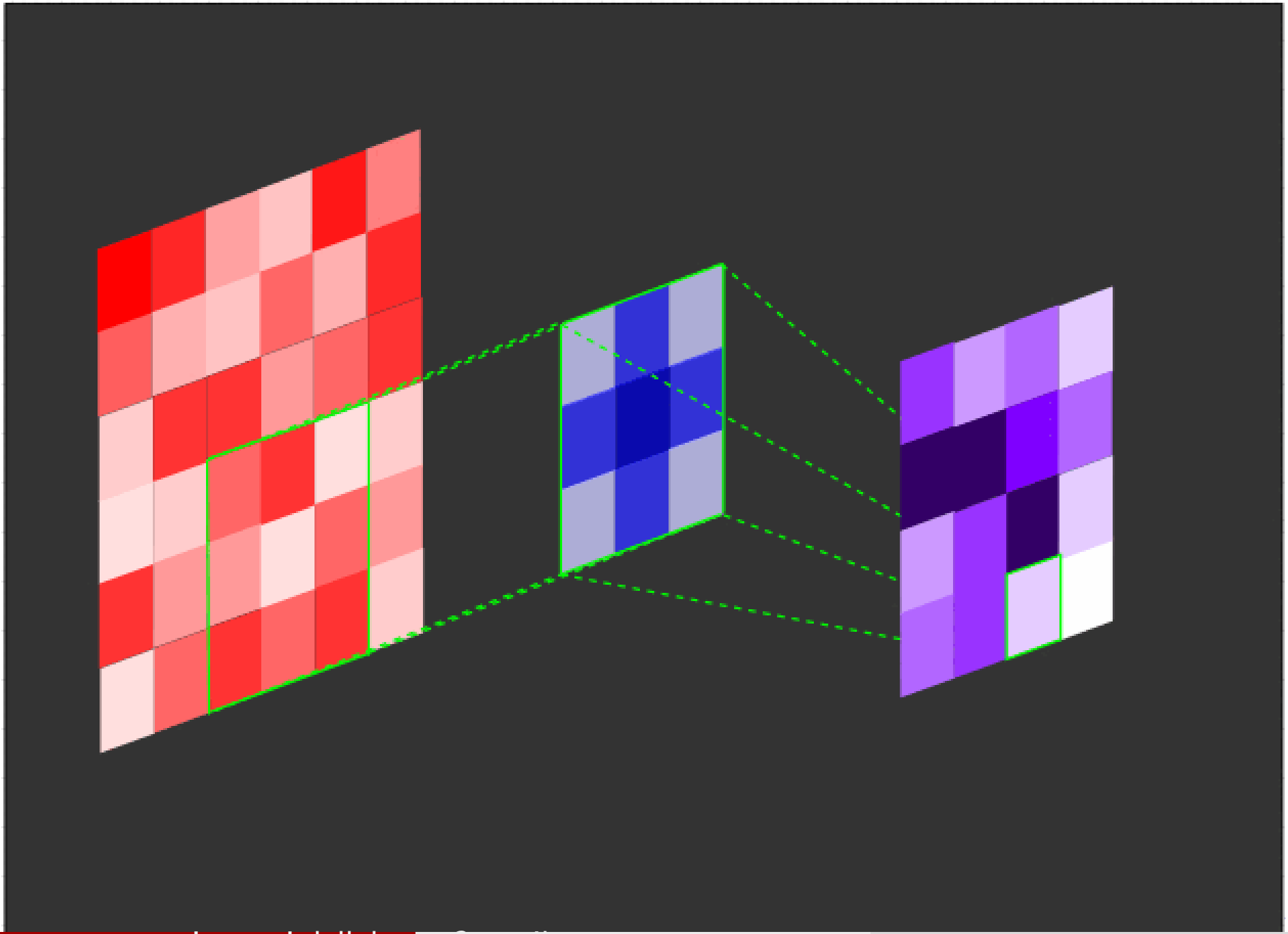


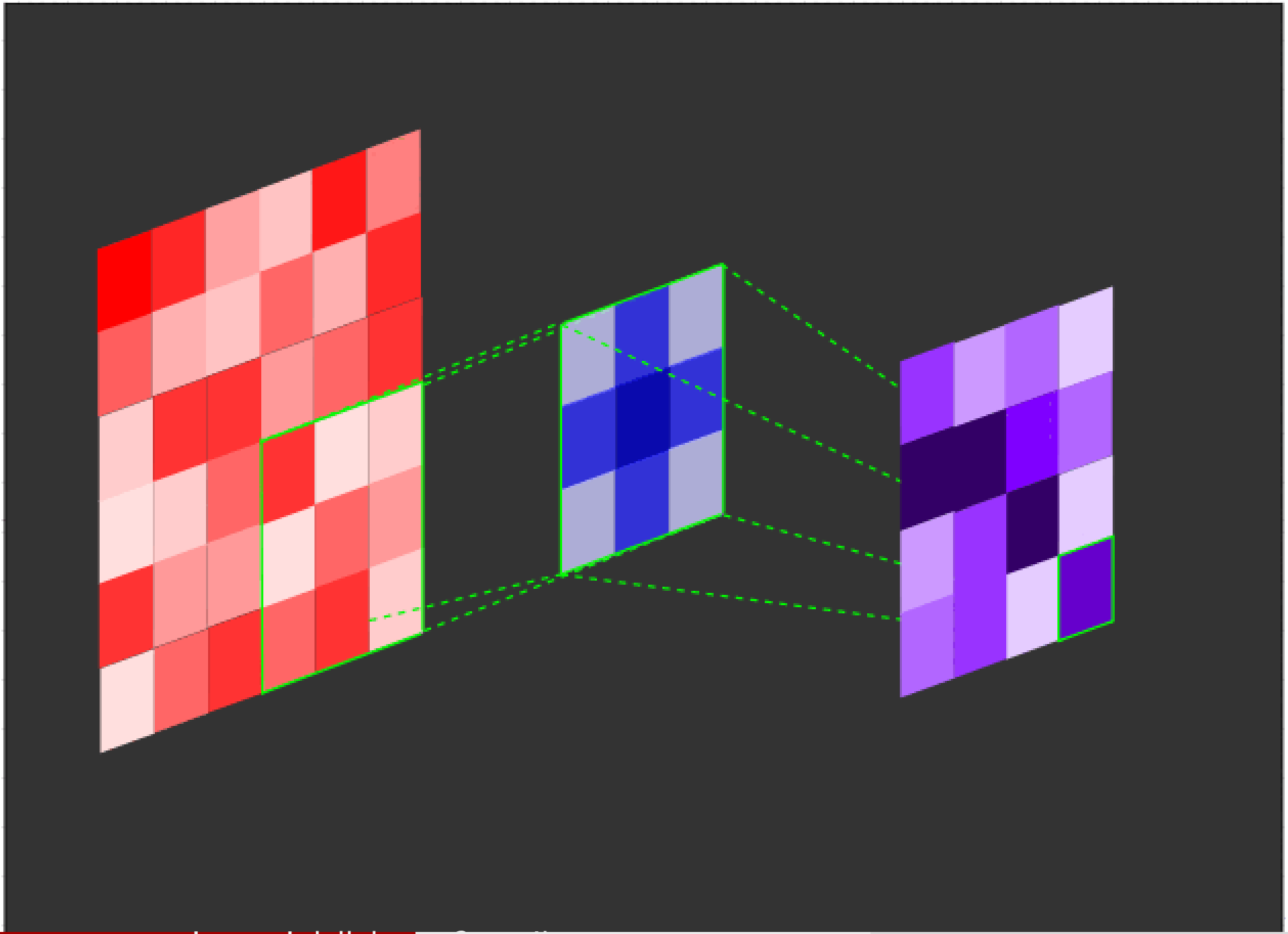


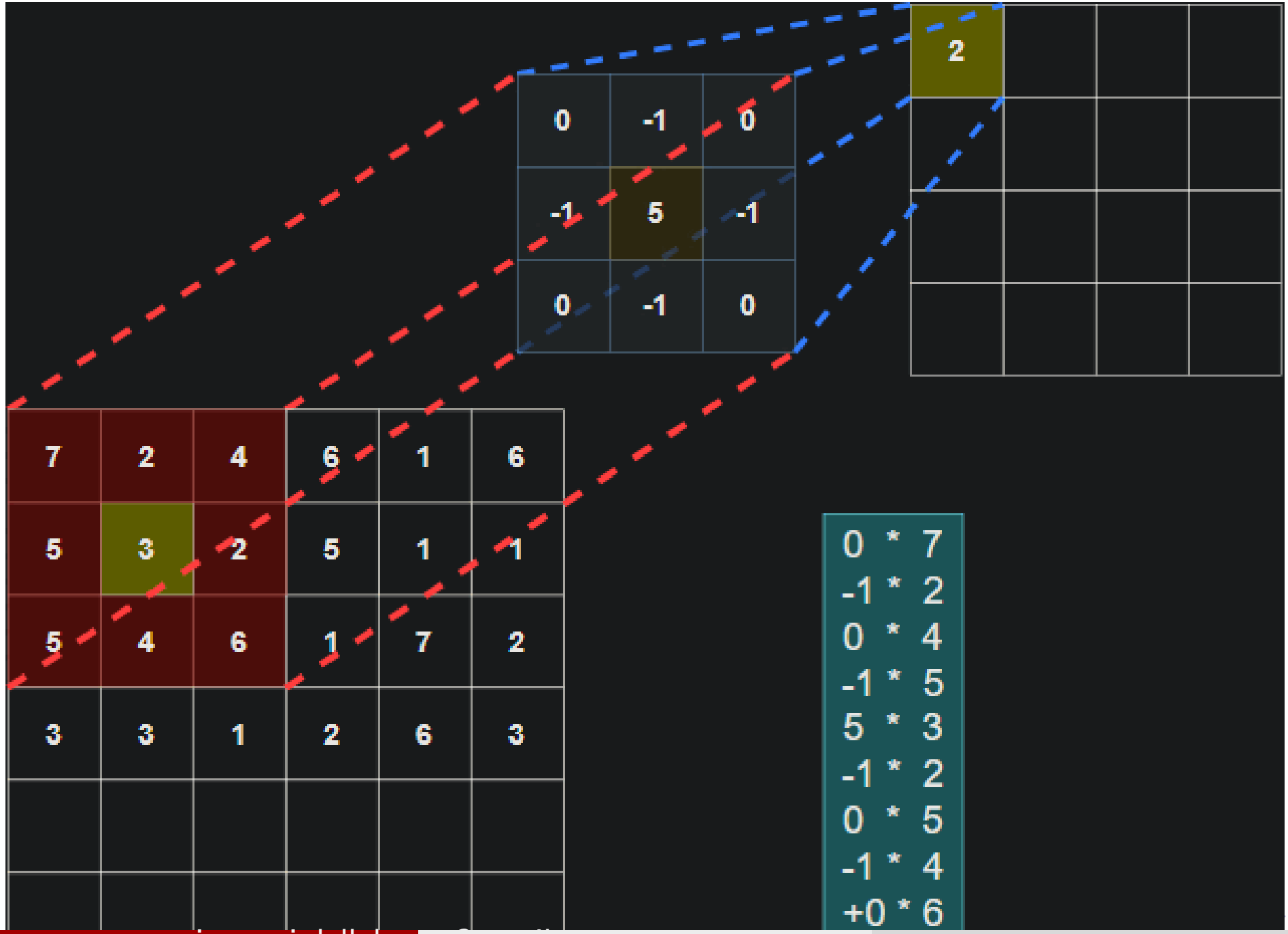


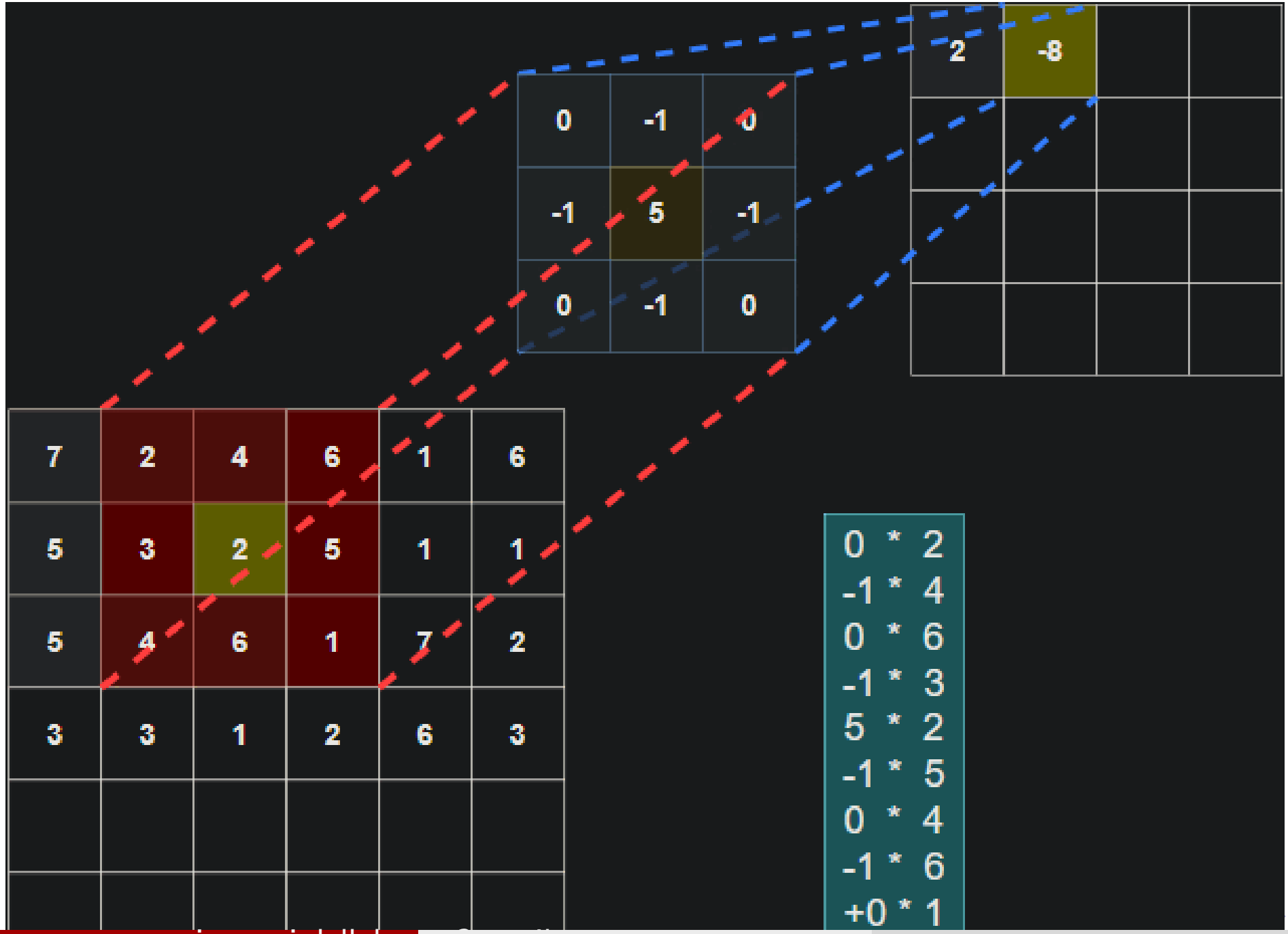


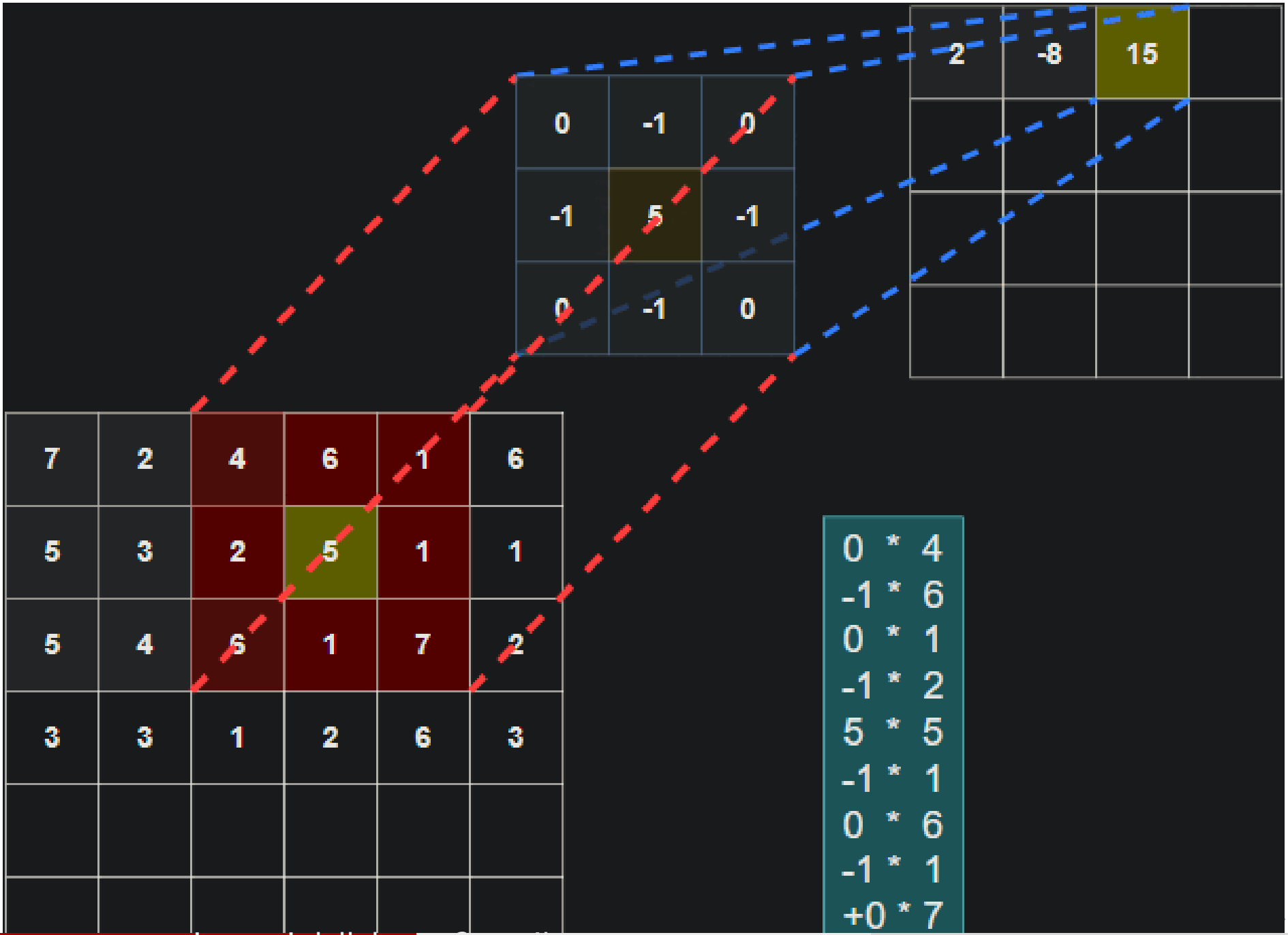


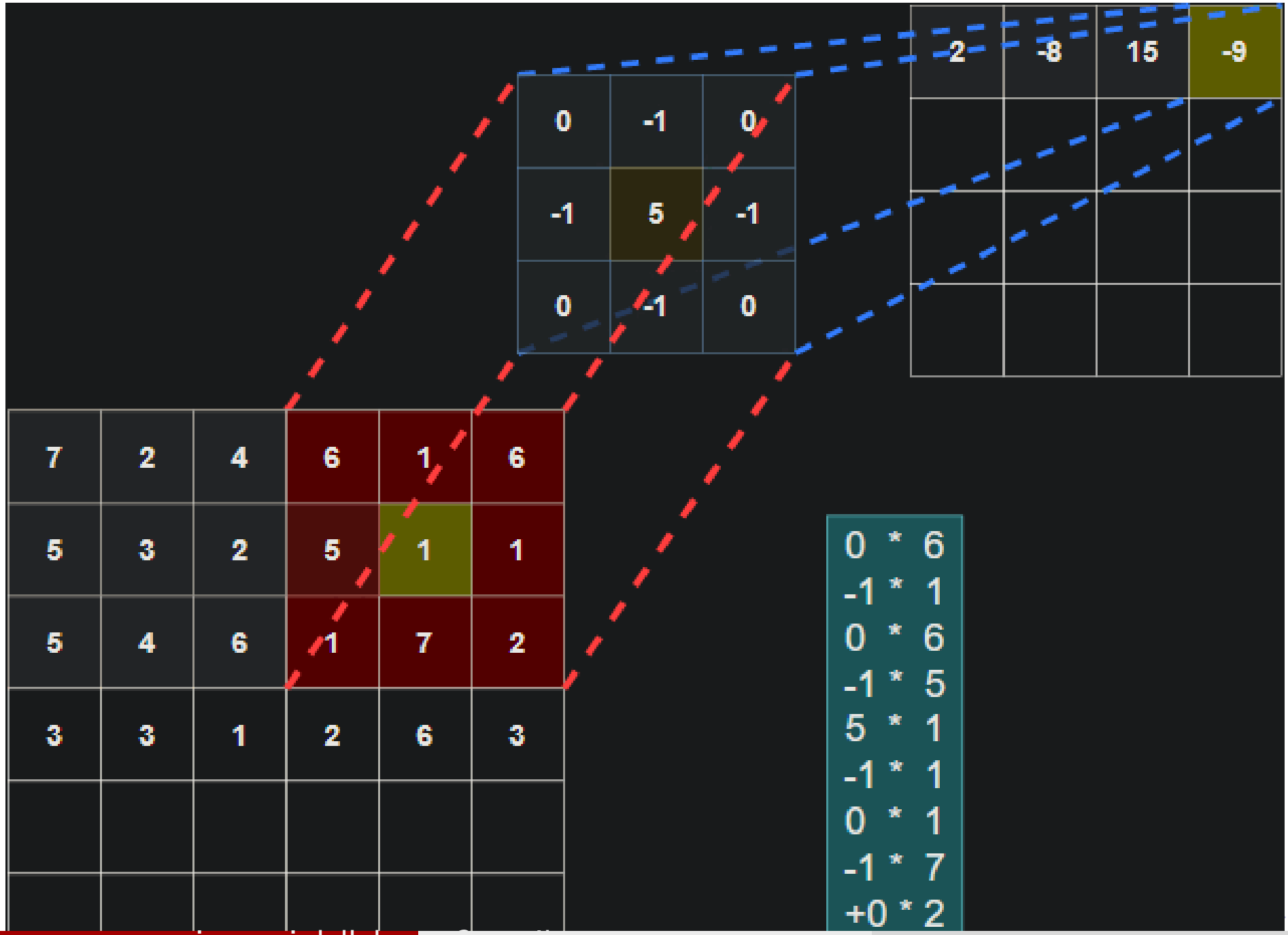





0 -1 0
-1 5 -1
0 -1 0
2
7 2 4 6 1 6
5 3 2 5 1 1
5 4 6 1 7 2
3 3 1 2 6 3
0 * 7
-1 * 2
0 * 4
-1 * 5
5 * 3
-1 * 2
0 * 5
-1 * 4
+0 * 6

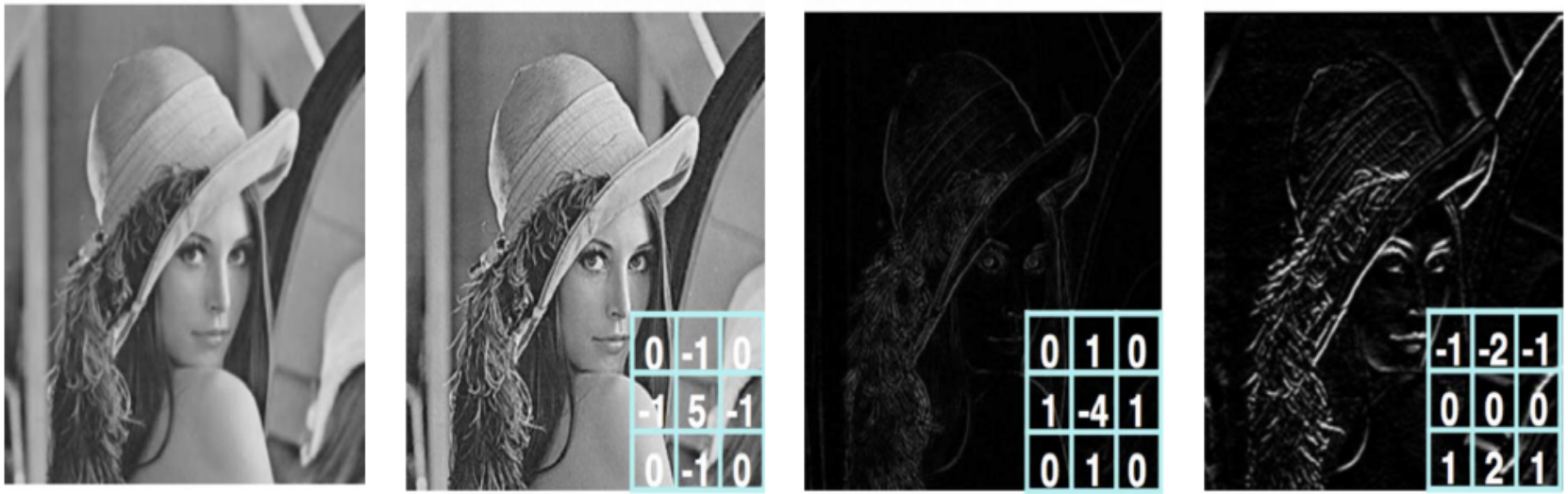

0 -1 0
-1 5 -1
0 -1 0
2 -8
7 2 4 6 1 6
5 3 2 5 1 1
5 4 6 1 7 2
3 3 1 2 6 3
0 * 2
-1 * 4
0 * 6
-1 * 3
5 * 2
-1 * 5
0 * 4
-1 * 6
+0 * 1


0 -1 0
-1 5 -1
0 -1 0
2 -8 15
7 2 4 6 1 6
5 3 2 5 1 1
5 4 6 1 7 2
3 3 1 2 6 3
0 * 4
-1 * 6
0 * 1
-1 * 2
5 * 5
-1 * 1
0 * 6
-1 * 1
+0 * 7


2 -8 15 -9
0 -1 0
-1 5 -1
0 -1 0
7 2 4 6 1 6
5 3 2 5 1 1
5 4 6 1 7 2
3 3 1 2 6 3
0 * 6
-1 * 1
0 * 6
-1 * 5
5 * 1
-1 * 1
0 * 1
-1 * 7
+0 * 2

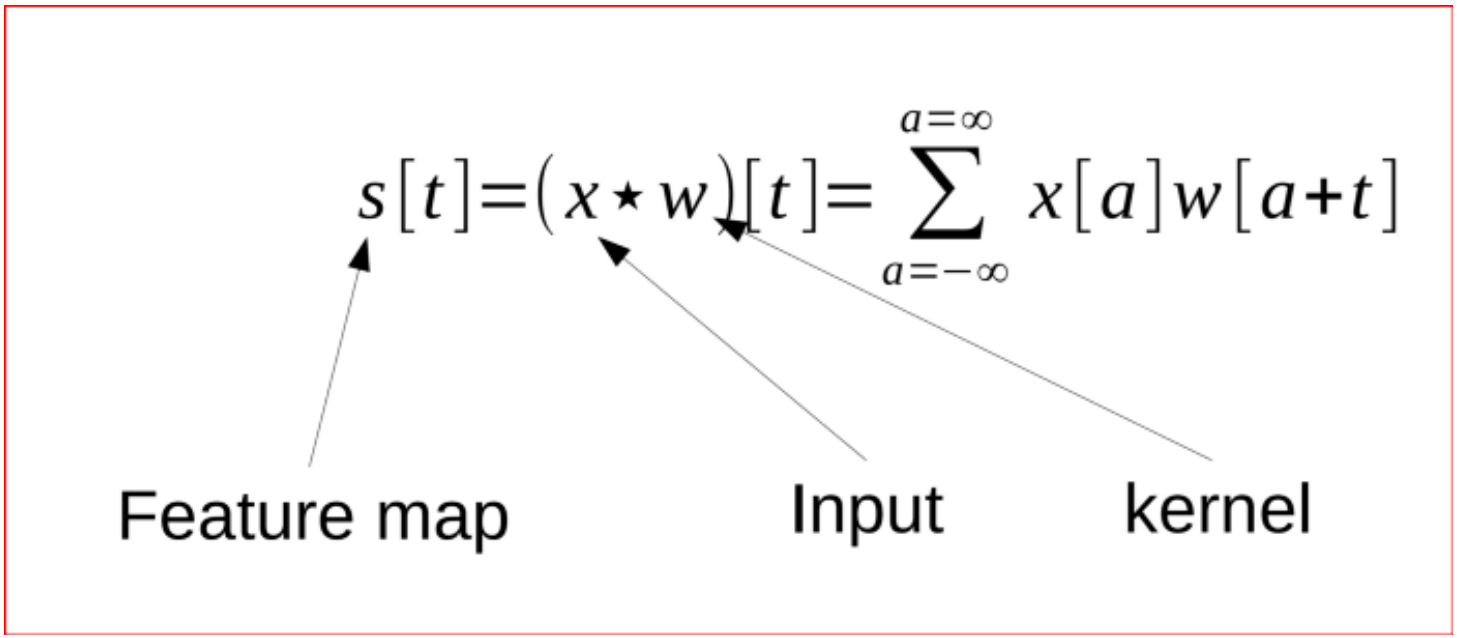
Producing Features Map

Convolutional layers typically have multiple kernels that are applied to the input image to extract different features. These features may include edges, textures, and patterns that are relevant to the given image recognition task.



Producing Features Map

Let's take an image **x**, which is a 2D array of pixels with different color channels(Red,Green and Blue-RGB). If we have a **feature detector or kernel w** then the output we get after applying the previous mathematical operation is called a **feature map**



Producing Features Map

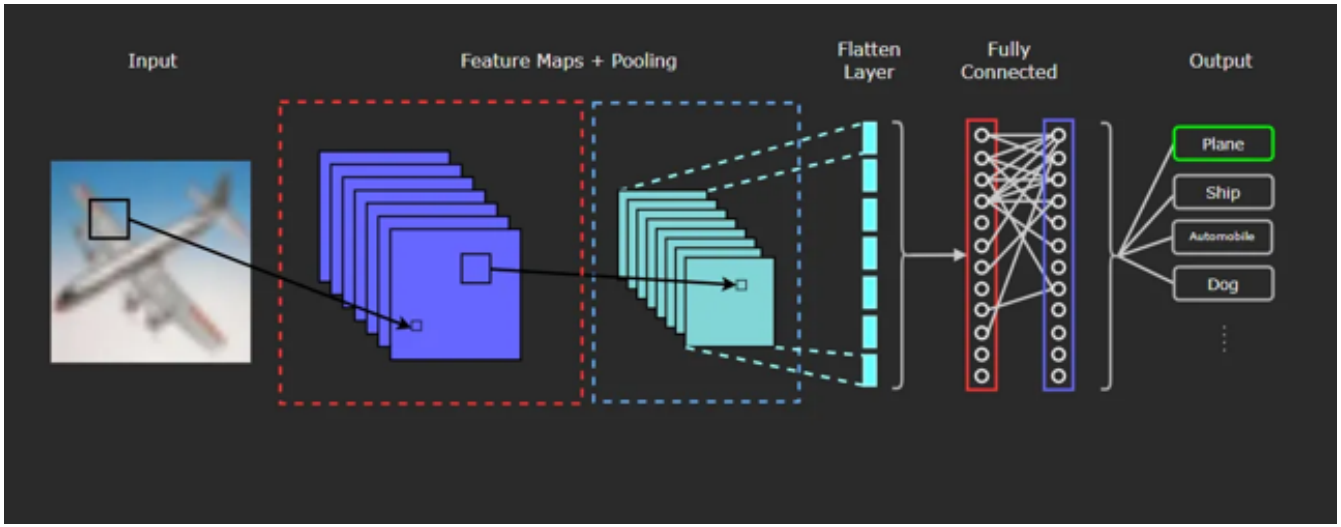
Finally, though the weights of the filters are the main parameters that are trained, there are also hyper-parameters that can be tuned for CNNs:

- number of filters in a layer
- dimensions of filters
- stride (number of pixels a filter moves each step)
- padding (how the filter handles boundaries of images)

We won't get into the details of these hyperparameters, since this isn't intended to be a comprehensive CNN walk-through, but these are important factors to be aware of.

CCN Architecture

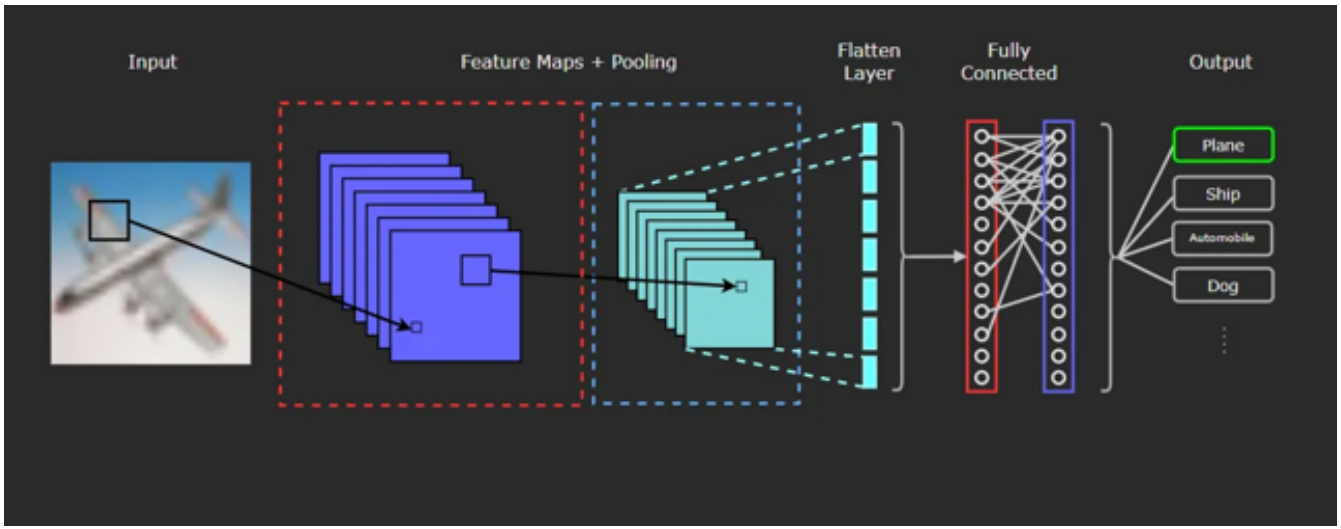
- The convolutional layer is only the beginning...
- In comparison to feed-forward networks, like the one we developed in the previous part of these lessons, CNN have different architecture, and **are composed of different types of layers**.
- In the figure below, we can see the general architecture of a typical CNN, including the different types of layers it can contain.



CCN Architecture

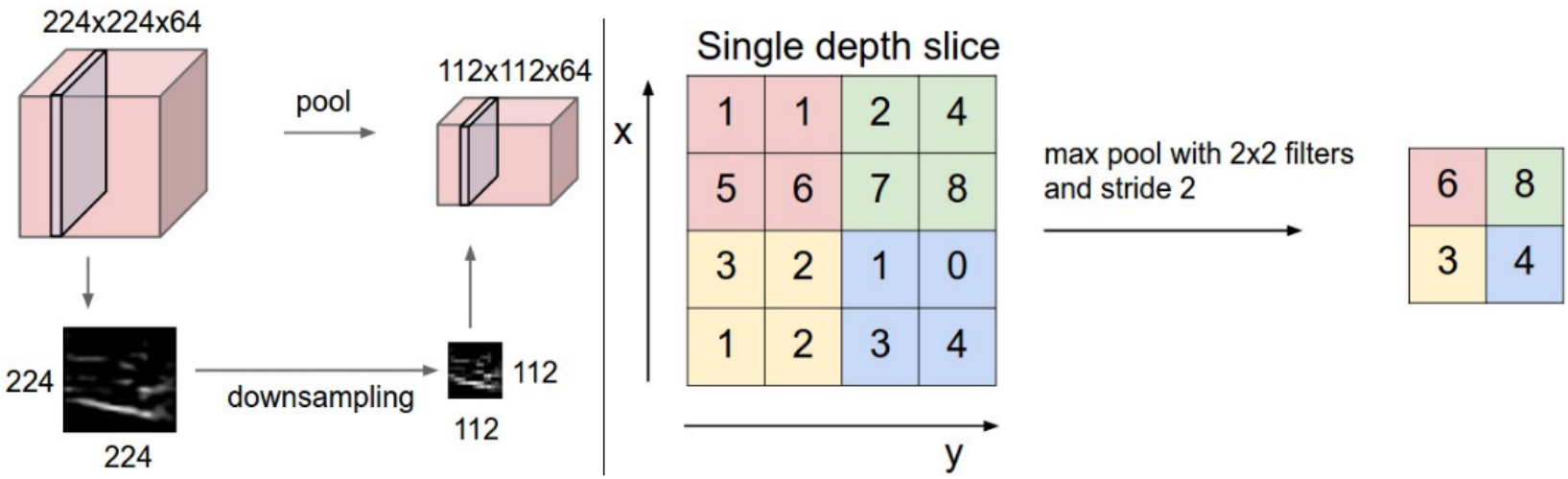
The three types of layers usually present in a Convolutional Network are:

- Convolutional Layers (red dashed outline)
- Pooling Layers (blue dashed outline)
- Fully Connected Layers (Red and Purple solid outlines)



Pooling

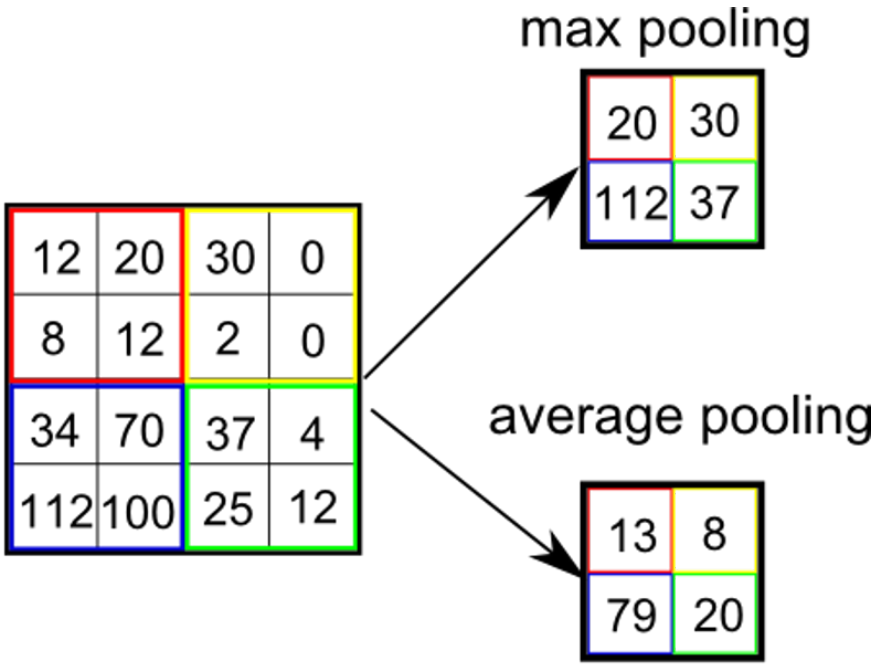
- Pooling layers are similar to convolutional layers, in that a filter convolves over the input data (usually a feature map that was output from a convolutional layer).
- However, rather than feature detection, the function of pooling layers is dimensionality reduction or downsampling.



Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. **Left:** In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. **Right:** The most common downsampling operation is max, giving rise to **max pooling**, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

Pooling

- The two most common types of pooling used are Max Pooling and Average Pooling.
- With Max Pooling, the filter slides across the input, and at each step will select the pixel with the largest value as the output.
- In Average Pooling, the filter will output the average value of the pixels that the filter is passing over.



CNN Output

- The last layer of a CNN is the classification layer which determines the predicted value based on the activation map.
- If you pass a handwriting sample to a CNN, the classification layer will tell you what letter is in the image.
- This is what autonomous vehicles use to determine whether an object is another car, a person, or some other obstacle.

Training

- Training a CNN is similar to training many other machine learning algorithms.
- You'll start with some training data that is separate from your test data and you'll tune your weights based on the accuracy of the predicted values.
- Just be careful that you don't overfit your model.

Subsection 2

Sequential Data

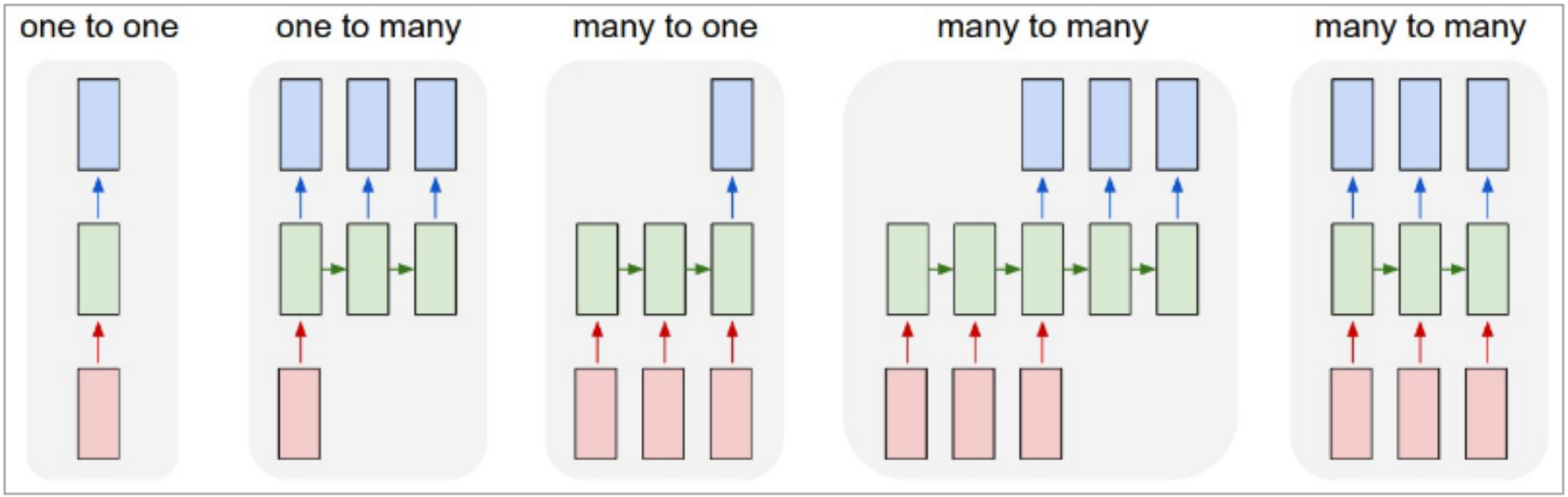
What are Sequential Data

- Sequential data refers to data that has a natural ordering to it, meaning that **each data point is dependent on the data that came before it**. In other words, **the sequence of the data matters**.
- Examples of sequential data include time series data such as stock prices, weather data, or audio signals, where each point in time is dependent on the previous point. Other examples include text data such as sentences or paragraphs, where the meaning of each word is dependent on the context of the words that came before it.
- Sequential data can be modeled and analyzed using various techniques, including Hidden Markov Models (HMMs), and autoregressive models such as ARIMA.

What are Sequential Data

- In the context of machine learning, sequential data presents unique challenges because the order of the data is important and traditional machine learning models such as decision trees and linear regression do not take this into account.
- Models such as RNNs (Recurrent Neural Network) are specifically designed to handle sequential data by processing the data one element at a time and retaining information about the previous elements to make predictions about future elements in the sequence.

What are Sequential Data



Each rectangle is a vector and arrows represent functions (e.g. matrix multiply). Input vectors are in red, output vectors are in blue and green vectors hold the RNN's state (more on this soon). From left to right: **(1)** Vanilla mode of processing without RNN, from fixed-sized input to fixed-sized output (e.g. image classification). **(2)** Sequence output (e.g. image captioning takes an image and outputs a sentence of words). **(3)** Sequence input (e.g. sentiment analysis where a given sentence is classified as expressing positive or negative sentiment). **(4)** Sequence input and sequence output (e.g. Machine Translation: an RNN reads a sentence in English and then outputs a sentence in French). **(5)** Synced sequence input and output (e.g. video classification where we wish to label each frame of the video). Notice that in every case are no pre-specified constraints on the lengths sequences because the recurrent transformation (green) is fixed and can be applied as many times as we like.

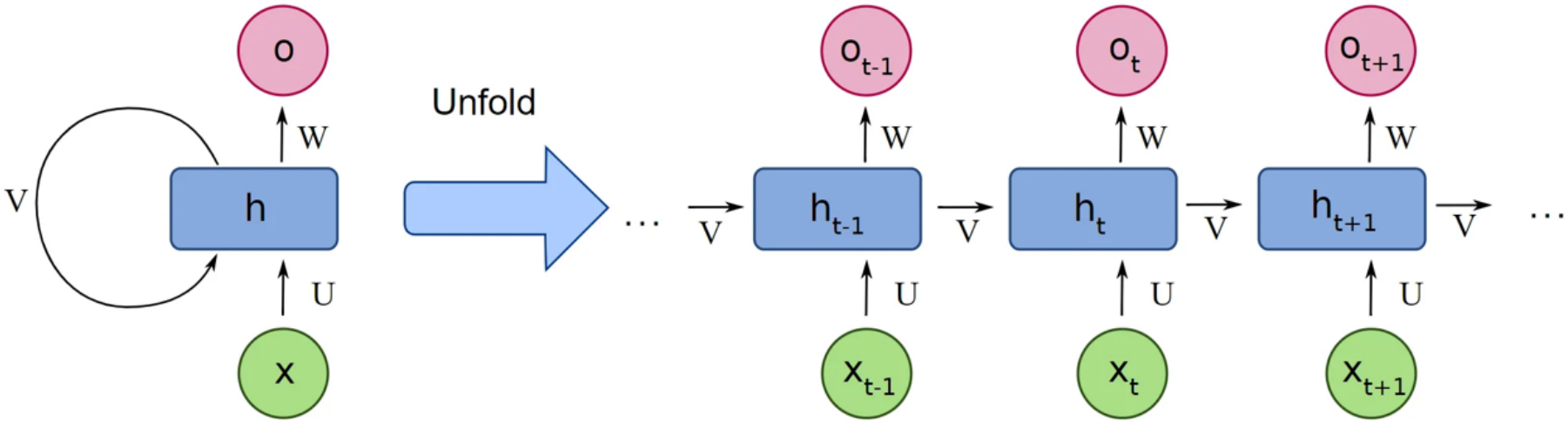
Subsection 3

Recurrent Neural Networks

What are Recurrent Neural Networks

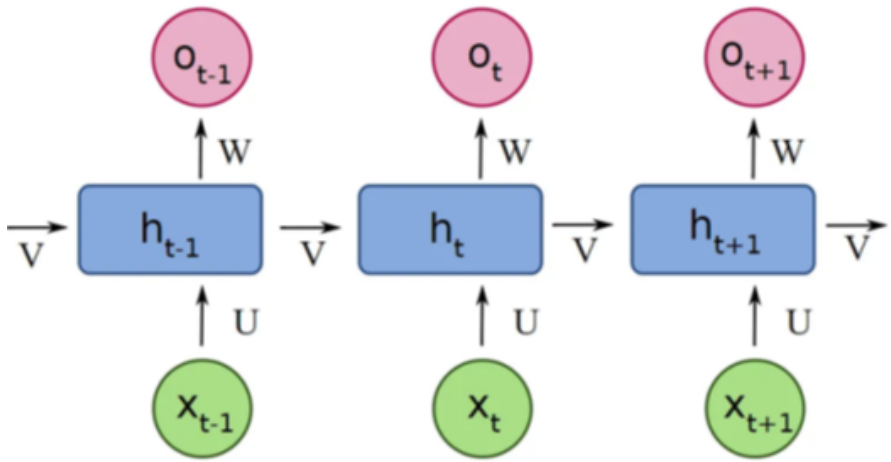
- A major characteristic of all neural networks you have seen so far, such as densely connected networks and convnets, is that they have no memory.
- Each input shown to them is processed independently, with no state kept in between inputs.

What are Recurrent Neural Network



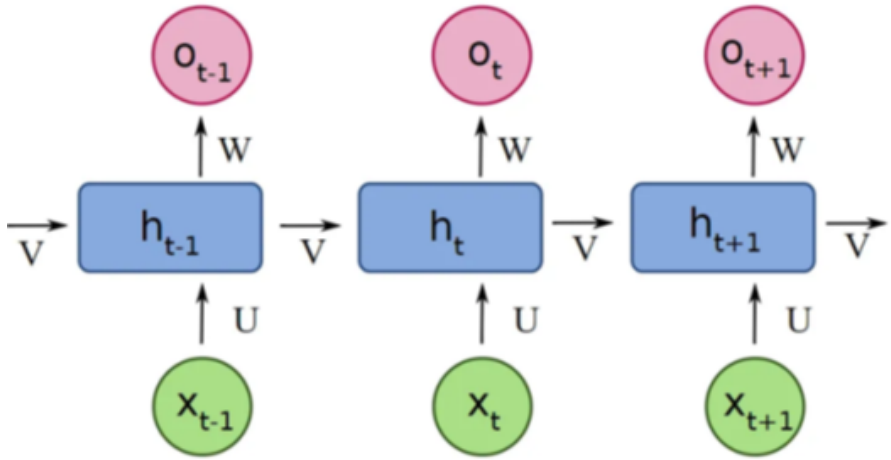
What are Recurrent Neural Network

- In the picture, there are some distinct components, from which the most important are:
- **x**: The input. It can be a word in a sentence or some other type of sequential data
- **O**: The output. For instance, what the network thinks the next word on a sentence should be given the previous words



What are Recurrent Neural Network

- In the picture, there are some distinct components, from which the most important are:
- **h**: The main block of the RNN. It contains the weights and the activation functions of the network
- **V**: Represents the communication from one time-step to the other.



What are Recurrent Neural Networks

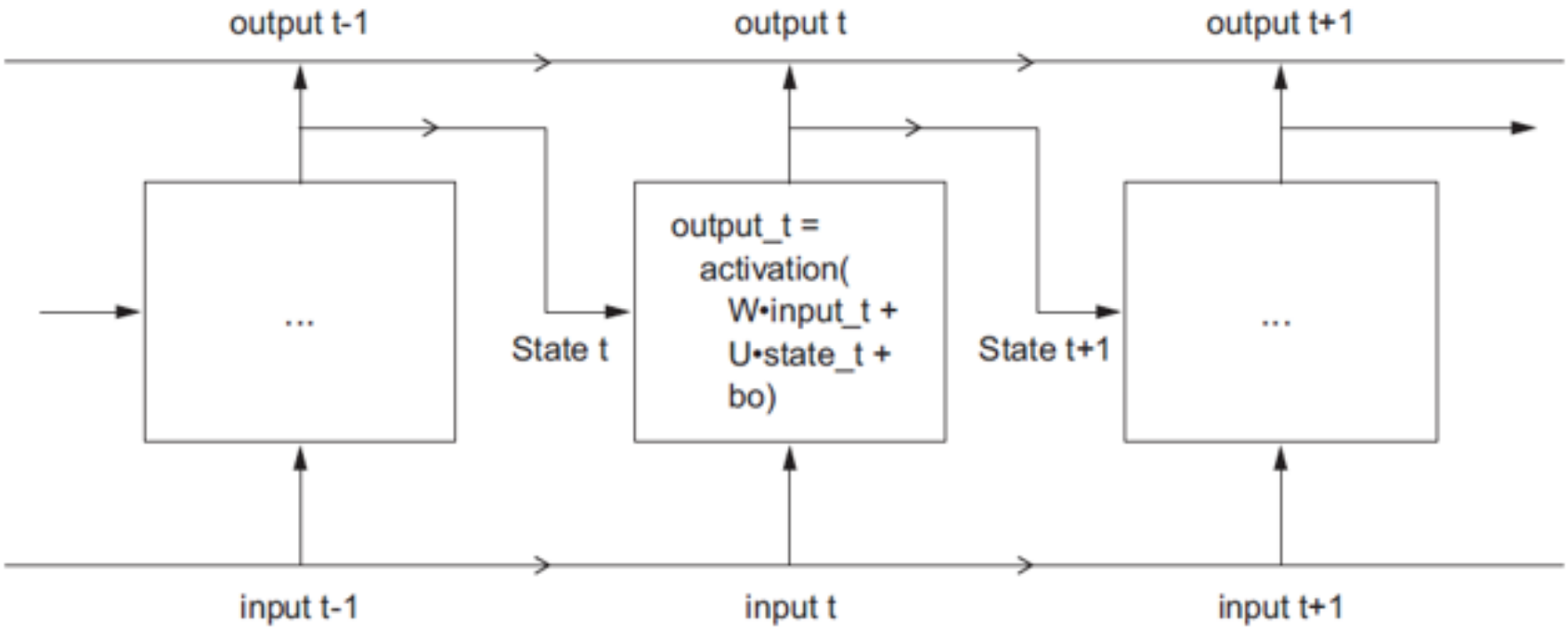


Figure 6.10 A simple RNN, unrolled over time

What are Recurrent Neural Networks

- A recurrent neural network (RNN) processes sequences by iterating through the sequence elements and maintaining a state containing information relative to what it has seen so far.
- In effect, an RNN is a type of neural network that has an internal loop: **the network internally loops over sequence elements**.

Listing 6.19 Pseudocode RNN

```
state_t = 0                          <-- The state at t
for input_t in input_sequence:       <-- Iterates over sequence elements
    output_t = f(input_t, state_t)
    state_t = output_t               <-- The previous output becomes the state for the next iteration.
```

What are Recurrent Neural Networks

- The transformation of the input and state into an output will be parameterized by two matrices, W and U, and a bias vector.
- It is similar to the transformation operated by a densely connected layer in a feedforward network.

Listing 6.20 More detailed pseudocode for the RNN

```
state_t = 0
for input_t in input_sequence:
    output_t = activation(dot(W, input_t) + dot(U, state_t) + b)
    state_t = output_t
```

What are Recurrent Neural Networks

- In this very simple example, the final output is a $2D$ tensor of shape $(timesteps, output\_features)$, where each timestep is the output of the loop at time t.
- Each timestep t in the output tensor contains information about timesteps 0 to t in the input sequence about the entire past.
- For this reason, in many cases, you don't need this full sequence of outputs; you just need the last output ($output\_t$ at the end of the loop), because it already contains information about the entire sequence.

Recurrent layer in Keras

The SimpleRNN layer:

```
from keras.layers import SimpleRNN
```

SimpleRNN processes batches of sequences, like all other Keras layers. This means it takes inputs of shape (*batch_size*, *timesteps*, *input_features*), rather than (*timesteps*, *Input_features*).

Subsection 4

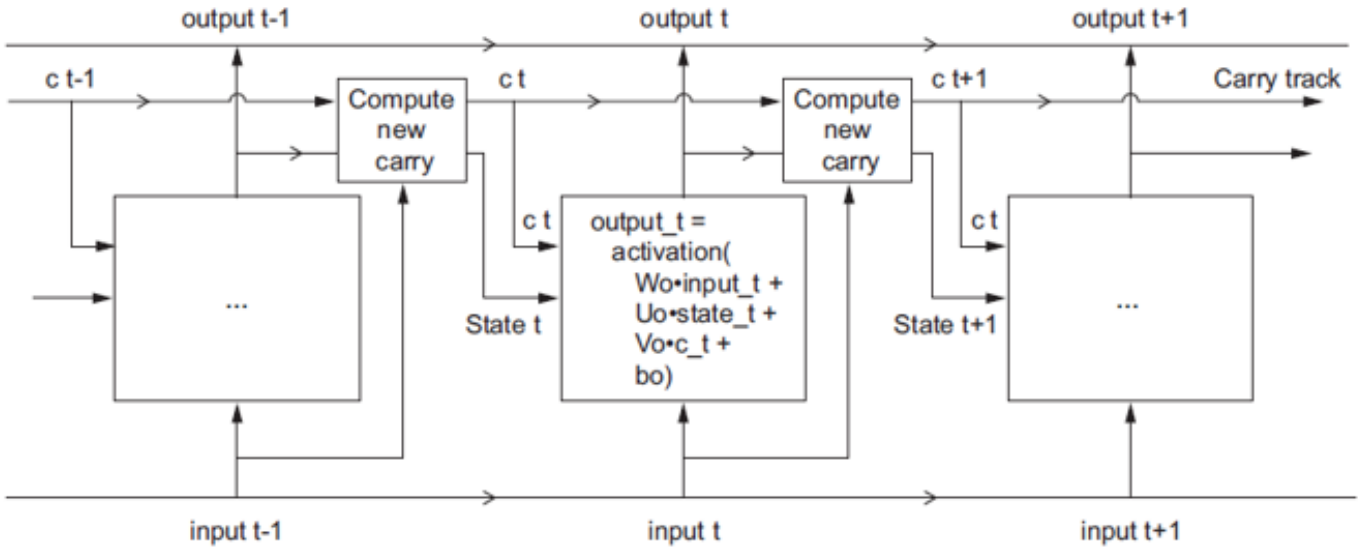
Long short-term memory NN

Problems with RNN

- SimpleRNN is not the only recurrent layer available in Keras.
- There are two others: **LSTM**and **GRU**.
- In practice, you will always use one of these, because SimpleRNN is generally too simplistic to be of real use.
- SimpleRNN has a major issue: although it should theoretically be able to retain at time t information about inputs seen many timesteps before, **in practice, such long-term dependencies are impossible to learn**.
- This is due to the **vanishing gradient problem**, an effect that is similar to what is observed with non-recurrent networks (feedforward networks) that are many layers deep: as you keep adding layers to a network, the network eventually becomes untrainable.
- The theoretical reasons for this effect were studied by Hochreiter, Schmidhuber, and Bengio in the early 1990s.
- The LSTM and GRU layers are designed to solve this problem.

LSTM layer in Keras

- Let's consider the LSTM layer.
- This layer is a variant of the SimpleRNN layer you already know about; it adds a way to carry information across many timesteps.



LSTM layer in Keras

Now the subtlety: the way the next value of the carry dataflow is computed. It involves three distinct transformations. All three have the form of a SimpleRNN cell:

```
y = activation(dot(state_t, U) + dot(input_t, W) + b)
```

Pseudocode details of the LSTM architecture

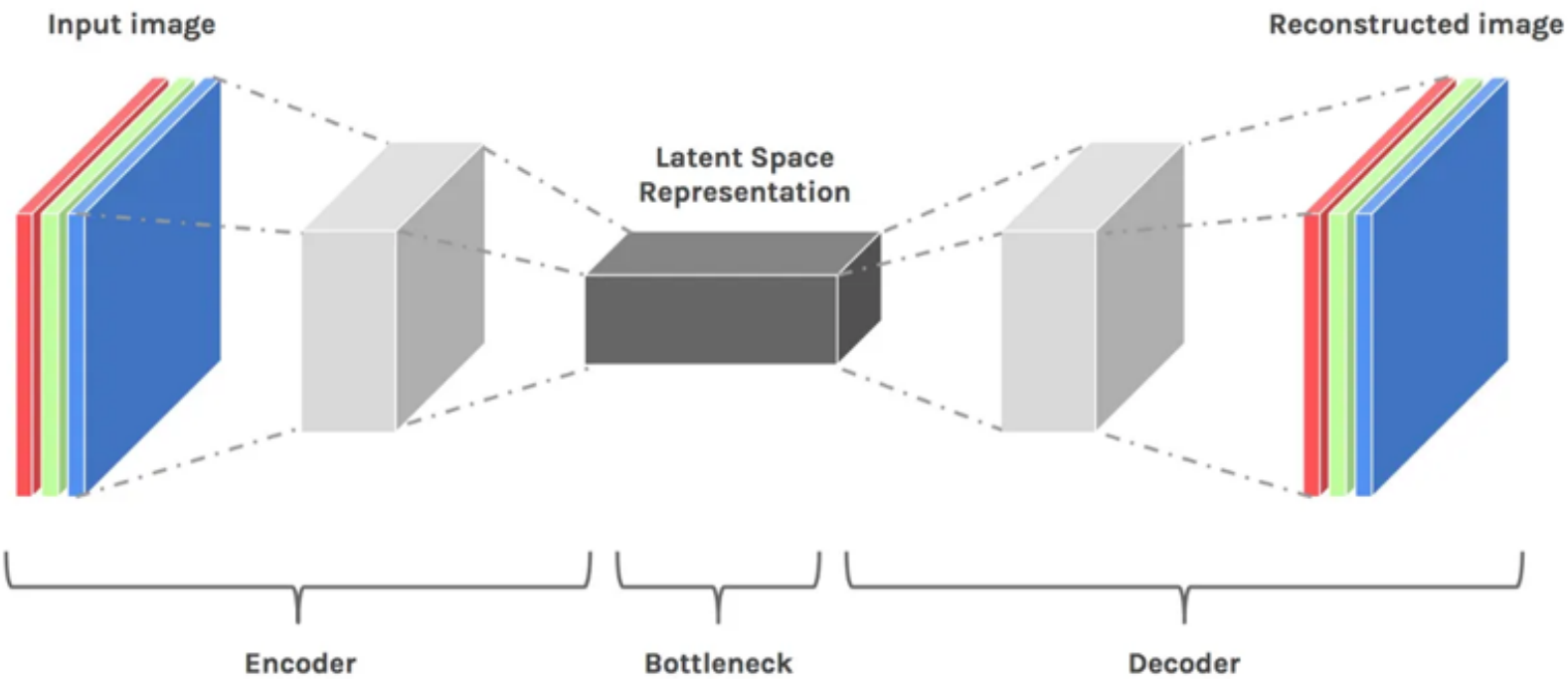
```
output_t = activation(dot(state_t, Uo) + \\
                      dot(input_t, Wo) + \\
                      dot(C_t, Vo) + bo)
i_t = activation(dot(state_t, Ui) + dot(input_t, Wi) + bi)
f_t = activation(dot(state_t, Uf) + dot(input_t, Wf) + bf)
k_t = activation(dot(state_t, Uk) + dot(input_t, Wk) + bk)
#
# You obtain the new carry state (the next c_t) by combining
# i_t, f_t, and k_t
#
c_t+1 = i_t * k_t + c_t * f_t
```

Subsection 5

Autoencoders

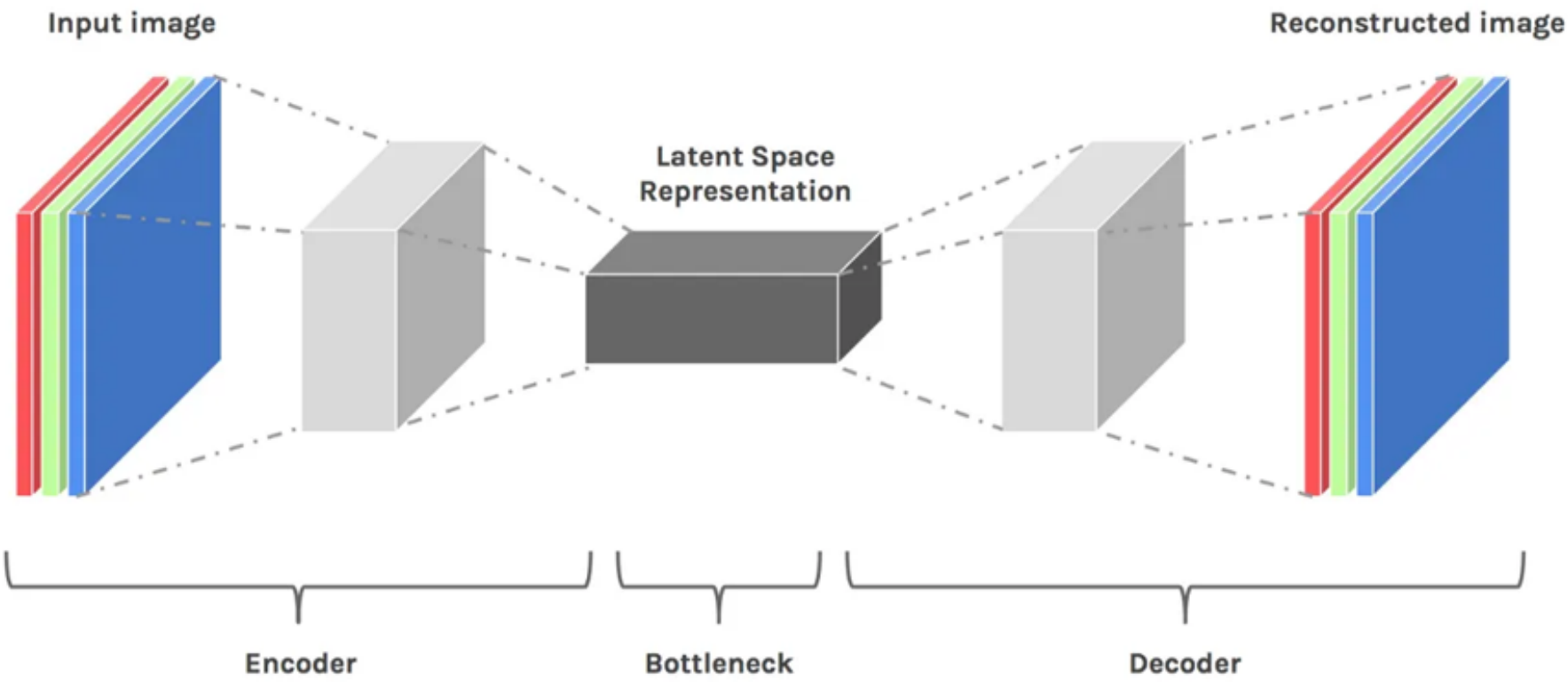
Autoencoders

- Autoencoders are a type of neural network that works in a self-supervised fashion.
- In autoencoders, there are three main building blocks: encoder, decoder, and coder or latent space. And then the decoder does the same but in the opposite order.



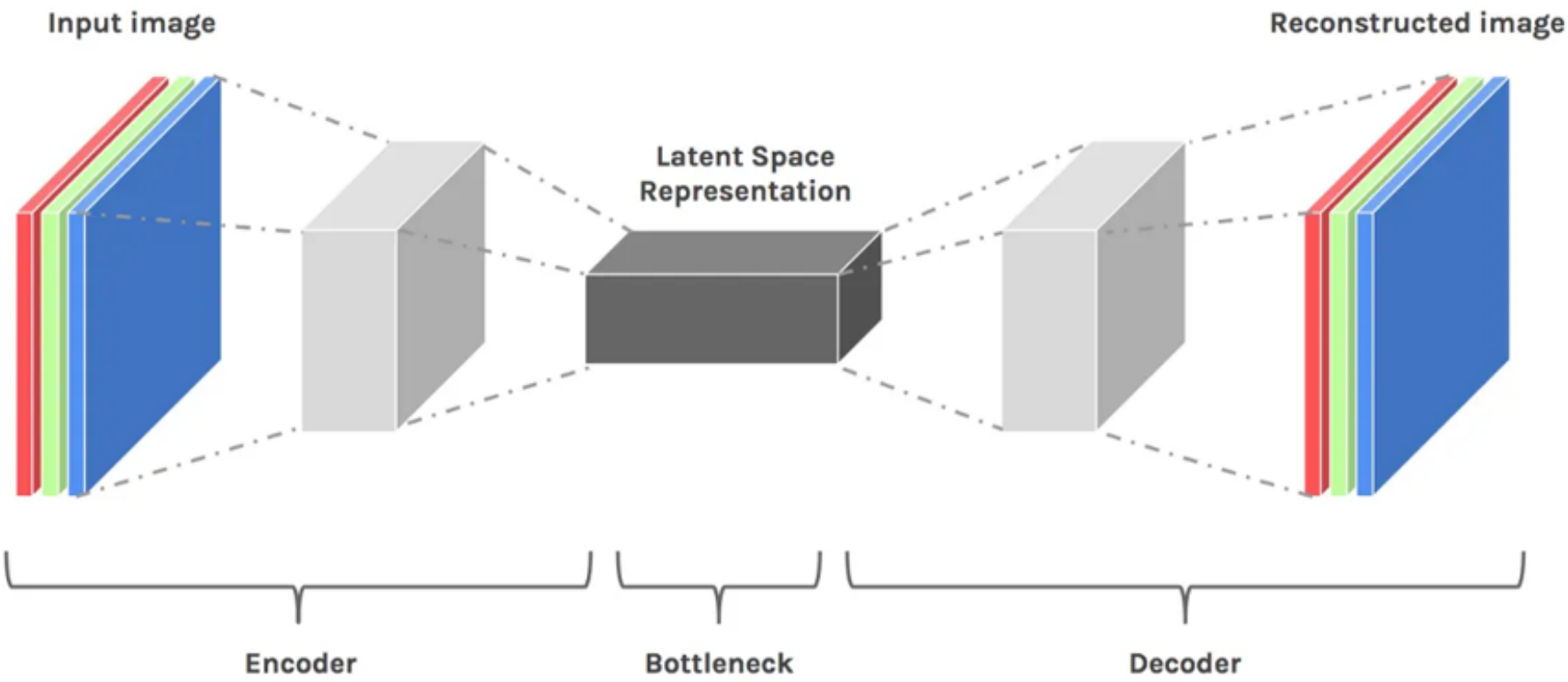
Autoencoders

As usual, you feed the autoencoder with data you want to use and then encoder **encodes or simply extracts useful features of input data and stores it in latent space**. And then the decoder does the same but in the opposite order.



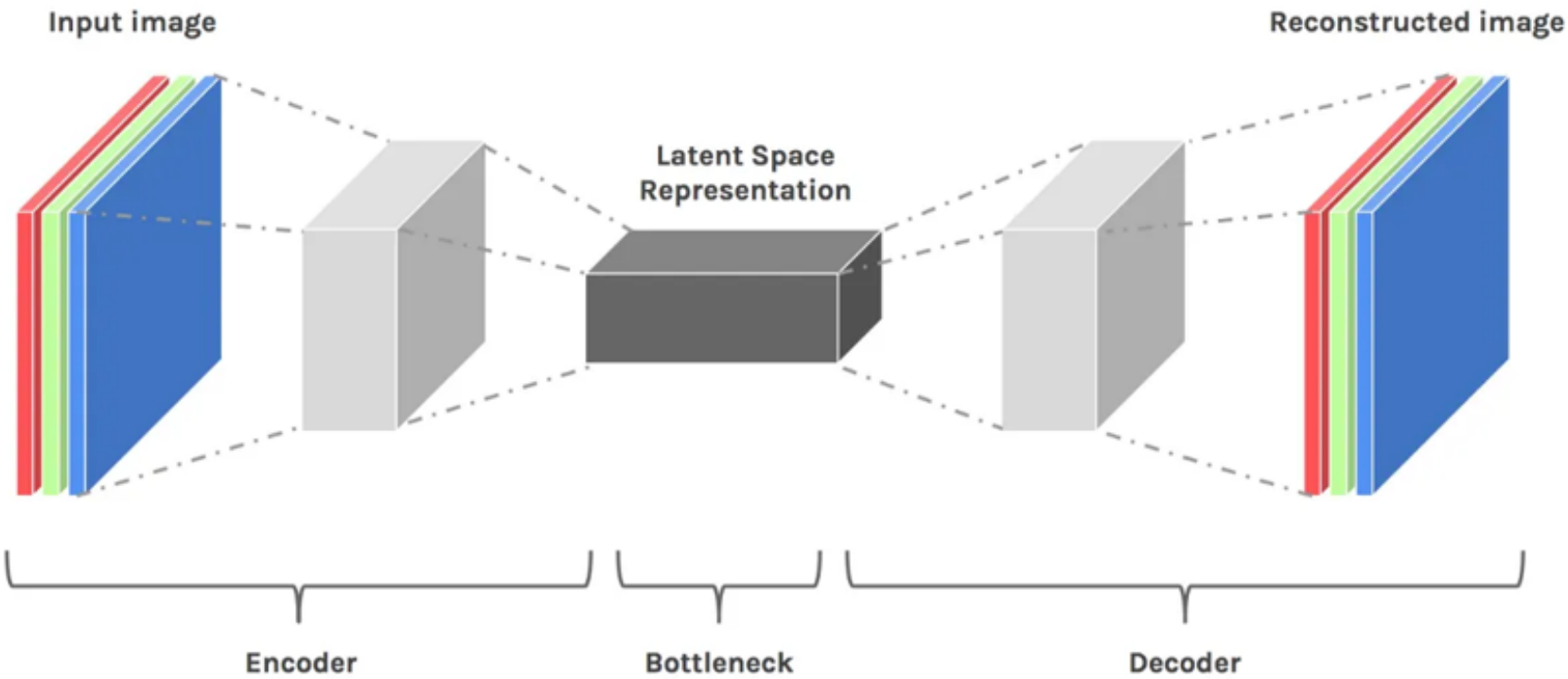
Autoencoders

As usual, you feed the autoencoder with data you want to use and then encoder **encodes or simply extracts useful features of input data and stores it in latent space**. And then the decoder does the same but in the opposite order.



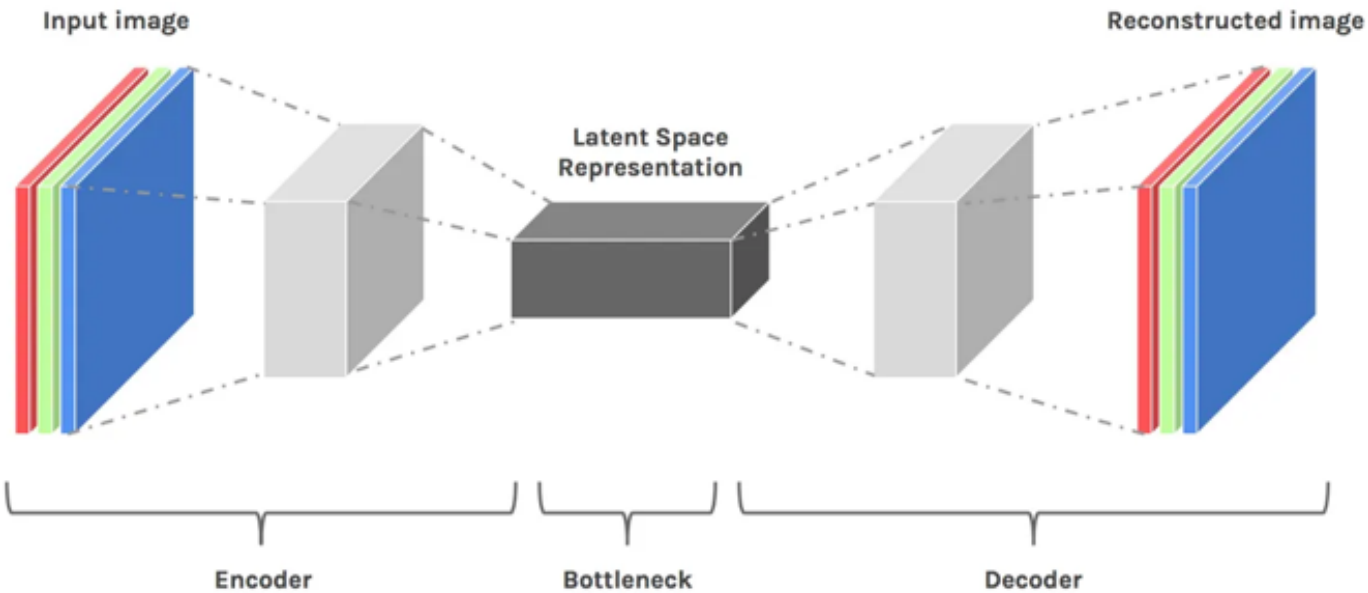
Autoencoders

- Data compression is one of the main advantages of autoencoders, it can be successfully implemented in data transmission problems.
- Imagine server uploads to internet only **Latent Space Representation** of the input image, depending on your request.



Autoencoders

- The main task of undercomplete autoencoders are not just copying input to the output, but instead, learn useful properties of input data.
- Autoencoder whose latent representation of input data dimension is less than the input dimension is called undercomplete.
- This type of autoencoder enables us **to capture the most notable features of the training data**.



Autoencoders

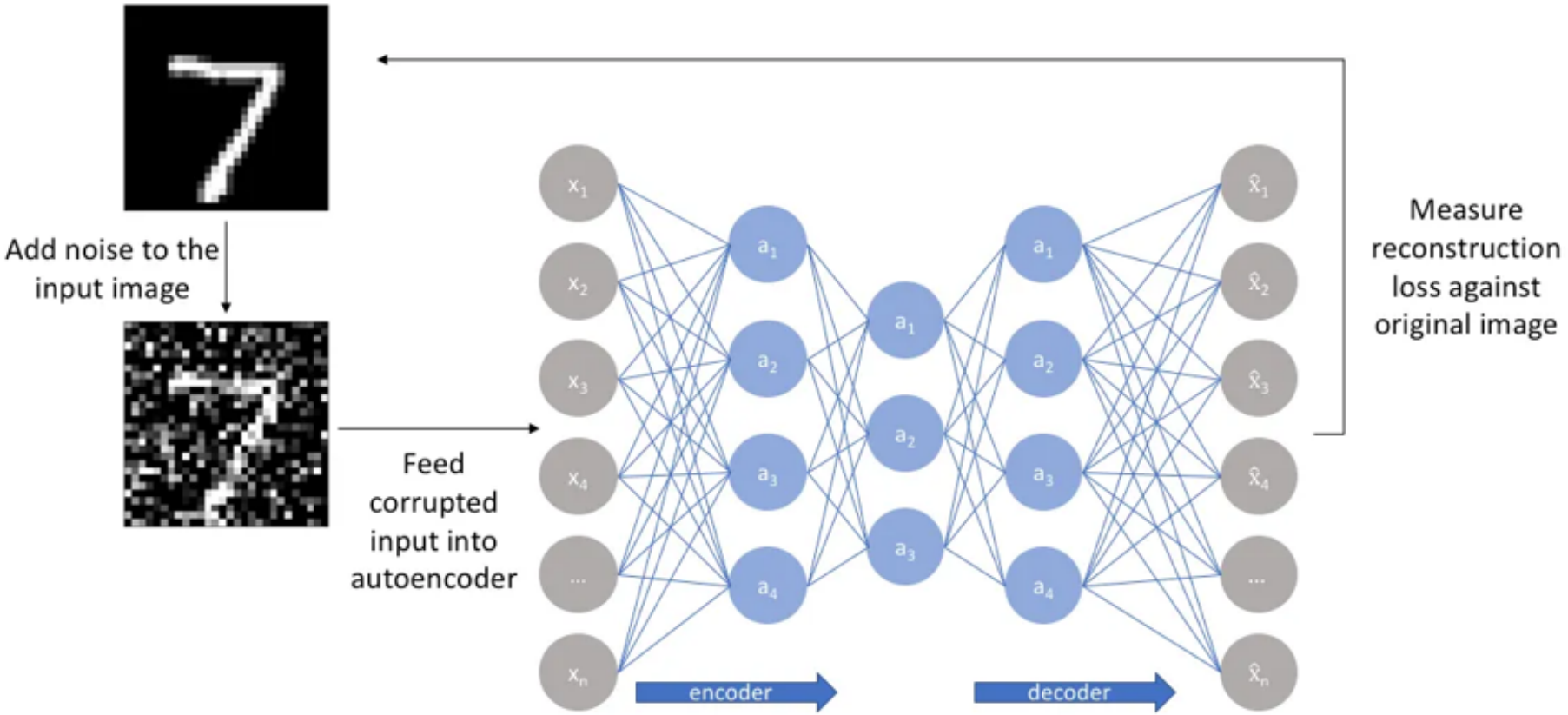
- **Data Reduction**
- By admitting that neural networks can learn nonlinear relationships in data, undercomplete autoencoders become a more powerful and nonlinear form of PCA for higher-dimensional data;
- Autoencoders are capable of learning complex representation of data, which can be then used in visualization in low-dimension space.

Autoencoders

- **Denoising Data**
- Quality loss of data (images/audio/video) is one main problem/issue in data transmission. Due to connection status or bandwidth, data such as image and audio can lose in quality, therefore the problem of denoising data arises.
- Denoising data is one of the cool advantages of autoencoders,
- Ideally, we would like to have our autoencoder is sensitive to reconstruct our latent space representation and insensitive enough to generalize the encoding of input data.
- One of the ways of such a method is to corrupt the input data, adding random noise to input data and then encode that corrupted data.

Autoencoders

Denoising Data

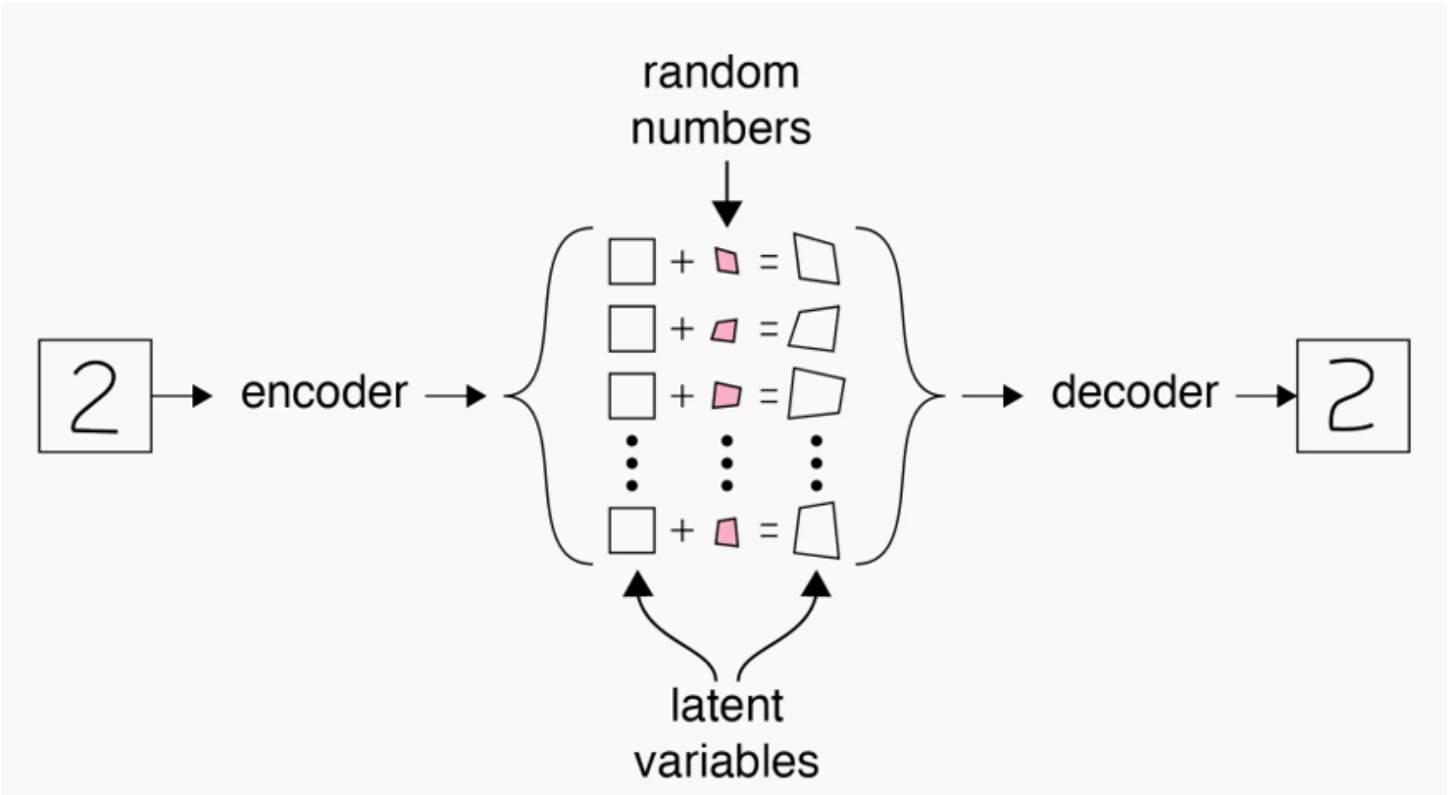


Variational Autoencoders

- VAEs inherit the architecture of traditional autoencoders and use this to learn a data generating distribution, which allows us to take random samples from the latent space.
- These random samples can then be decoded using the decoder network to generate unique data that have similar characteristics to those that the network was trained on.

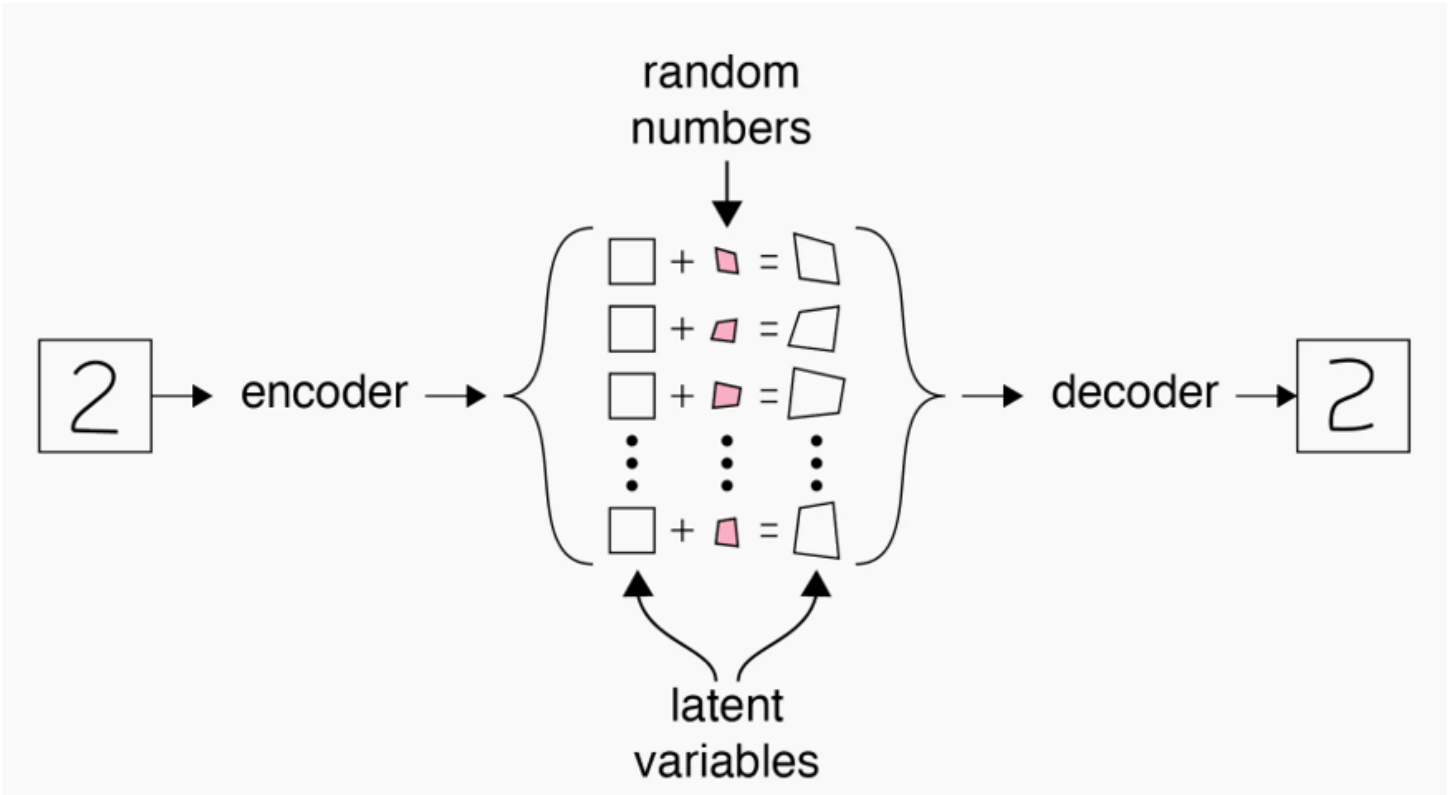
Variational Autoencoders

- **The Data Generating Procedure**
- Looking at the below image, we can consider that our approximation to the data generating procedure decides that we want to generate the number '2', so it generates the value 2 from the latent variable centroid.



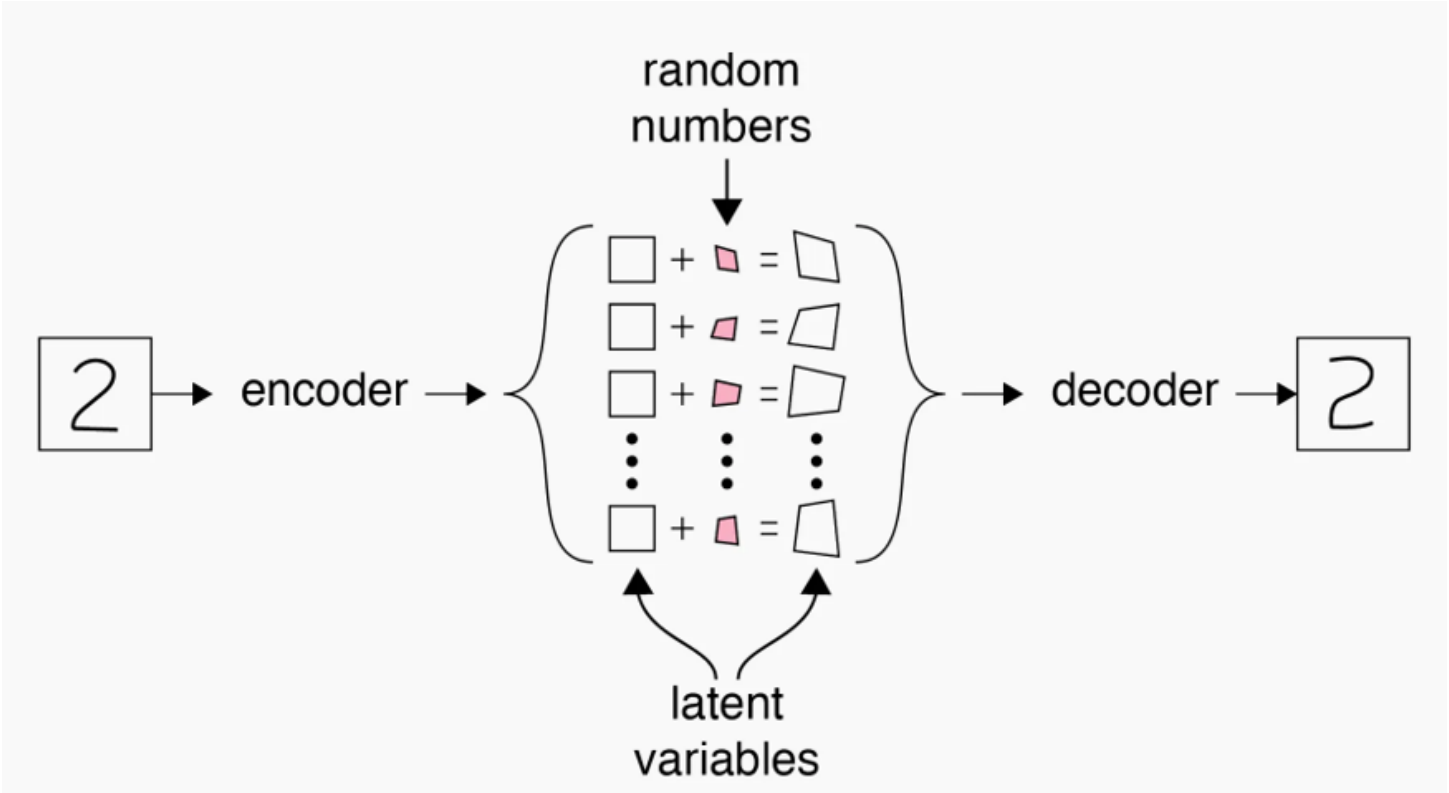
Variational Autoencoders

- **The Data Generating Procedure**
- However, we may not want to generate the same looking ‘2‘ every time so we add some random noise to this item in the latent space, which is based on a random number and the ‘learned‘ spread of the distribution for the value ‘2‘.



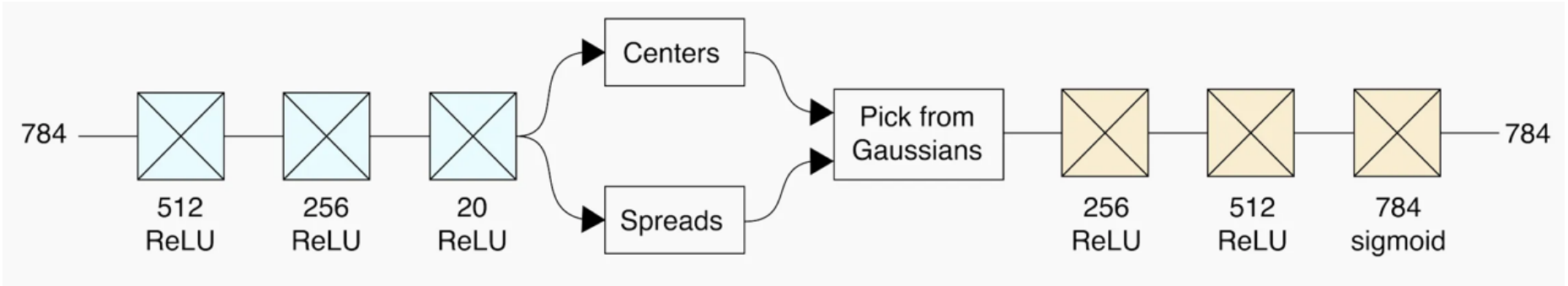
Variational Autoencoders

- **The Data Generating Procedure**
- We pass this through our decoder network and we get a 2 which looks different to the original.



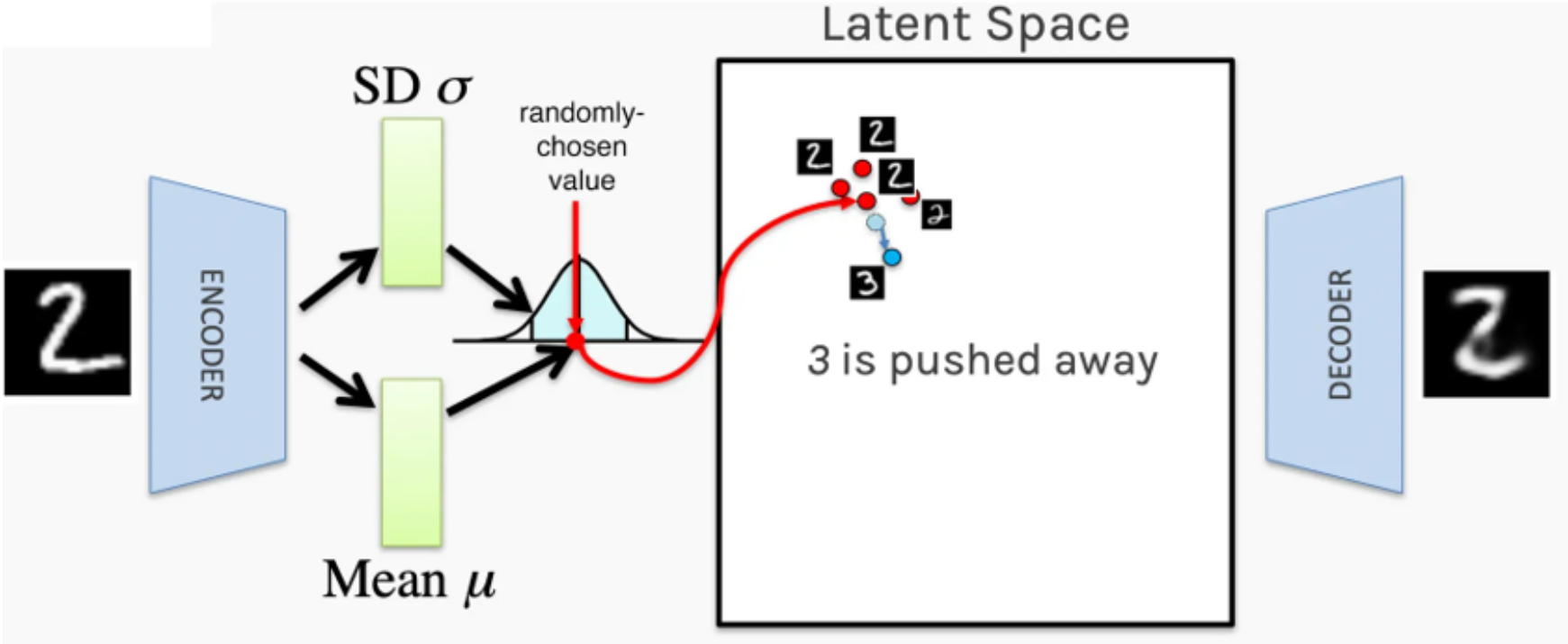
Variational Autoencoders

- The previous one was an oversimplified version which abstracted the architecture of the actual autoencoder network.
- Below is a representation of the architecture of a real variational autoencoder using convolutional layers in the encoder and decoder networks.
- We see that we are learning the centers and spreads of the data generating distributions within the latent space separately, and then ‘sampling‘ from these distributions to generate essentially ‘fake‘ data.



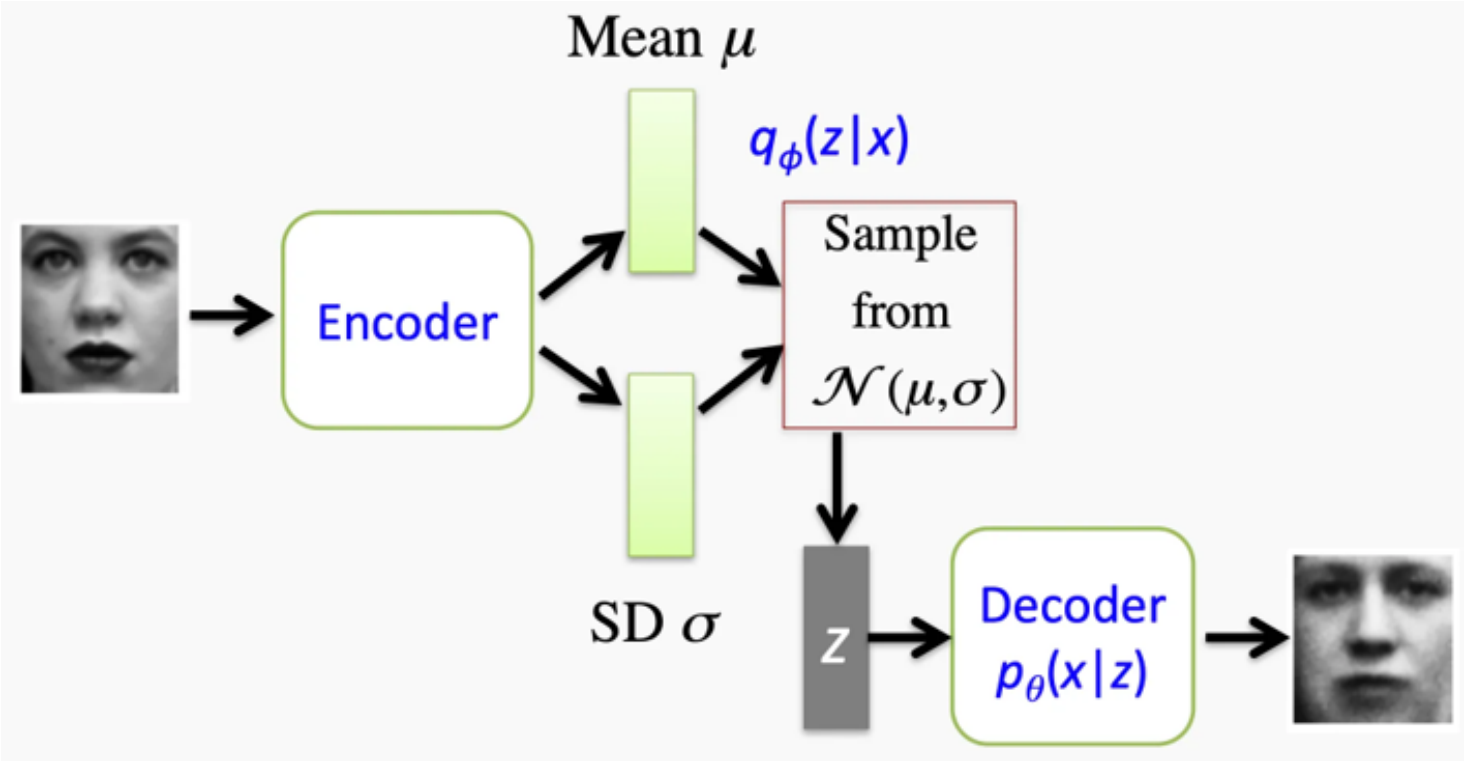
Variational Autoencoders

The inherent nature of the learning procedure means that parameters that look similar (stimulate the same network neurons to fire) are clustered together in the latent space, and are not spaced arbitrarily. This is illustrated in the figure below. We see that our values of 2's begin to cluster together, whilst the value 3 gradually becomes pushed away. This is useful as it means the network does not arbitrarily place characters in the latent space, making the transitions between values less spurious.



Variational Autoencoders

We train the autoencoder using a set of images to learn our mean and standard deviations within the latent space, which forms our data generating distribution. Next, when we want to generate a similar image, we sample from one of the centroids within the latent space, distort it slightly using our standard deviation and some random error, and then pass this through the decoder network. It is clear from this example that the final output looks similar, but not the same, as the input image.



Subsection 6

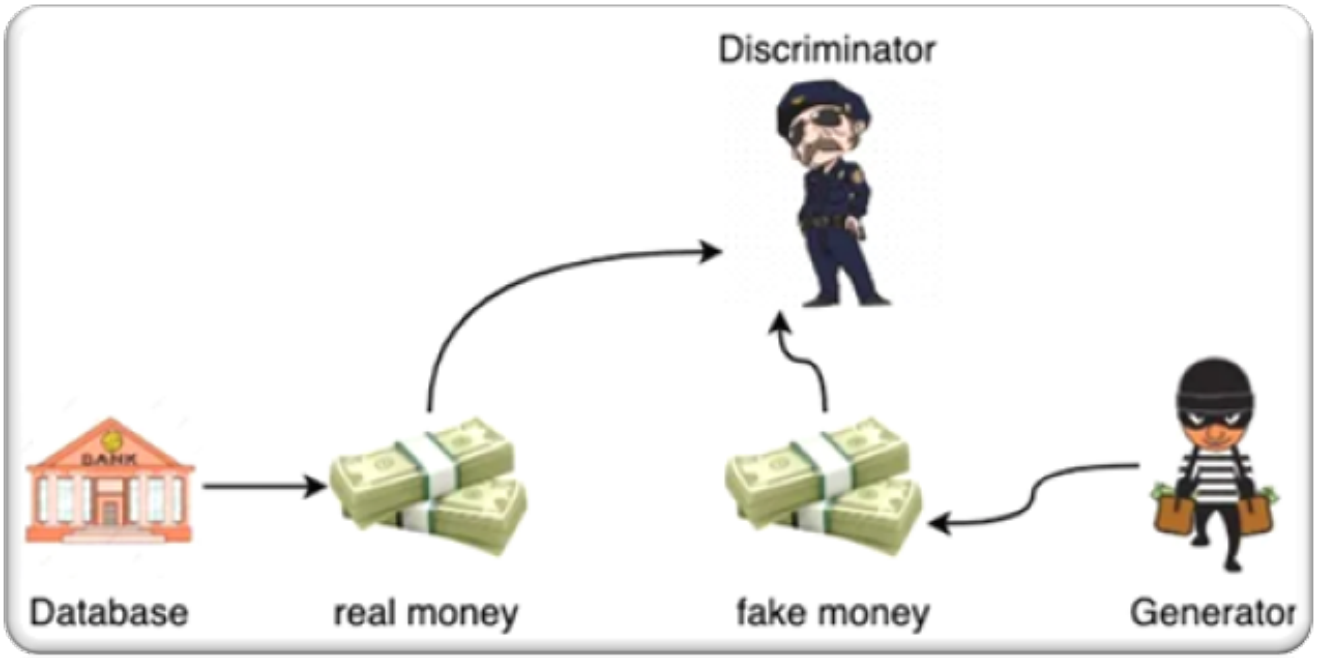
Generative Adversarial Networks GAN

Generative Adversarial Networks

- A generative adversarial network (GAN) is a type of artificial intelligence (AI) algorithm that can learn to generate new data that is similar to a training set of data.
- It works by training two neural networks, a **generator** and a **discriminator**, to compete against each other in a zero-sum game.
- The generator creates new data samples based on random noise, while the discriminator tries to distinguish between the generated samples and real ones from the training set.
- As the generator gets better at creating realistic samples, the discriminator gets better at distinguishing them from the real ones.

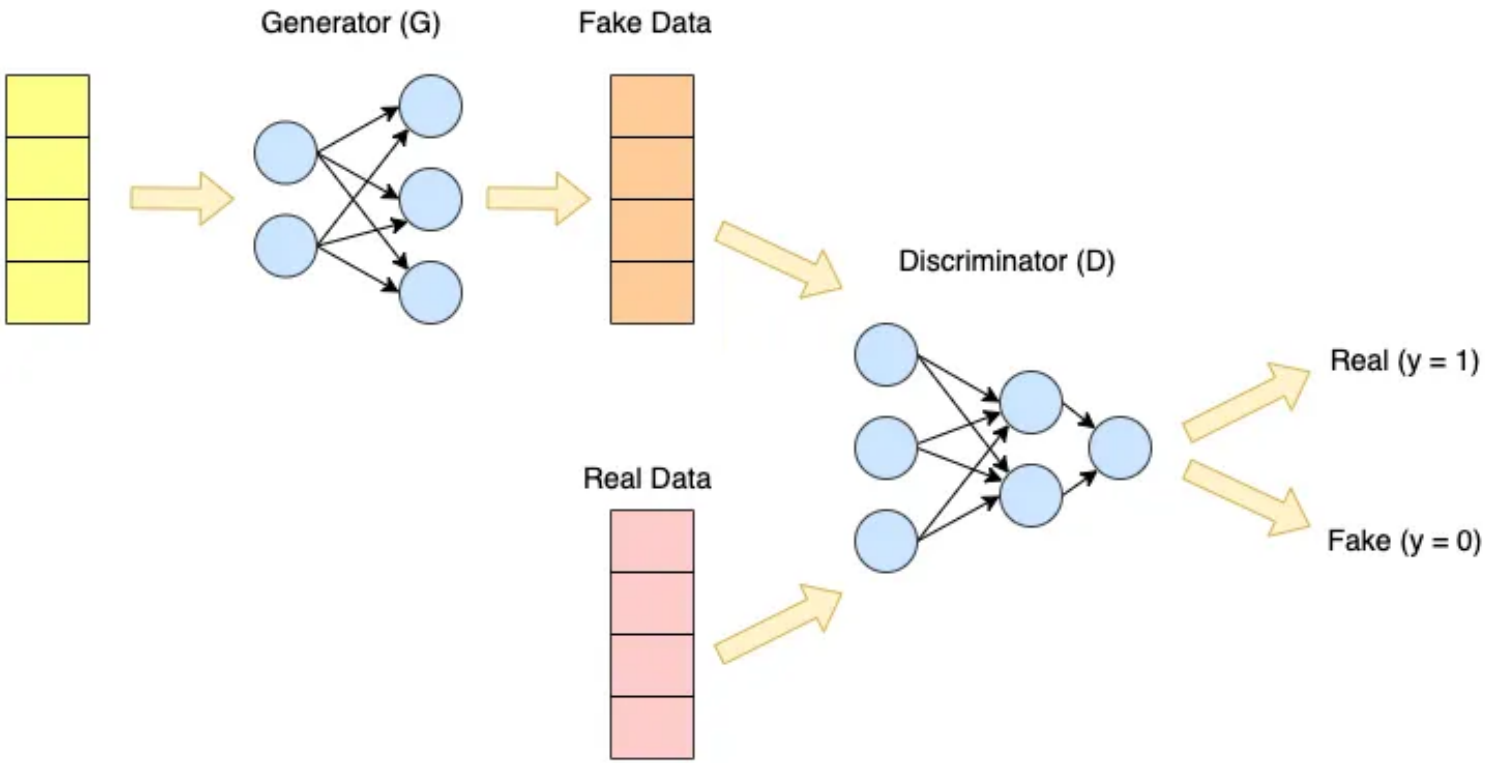
Generative Adversarial Networks

- Through this process of competition and feedback, the generator learns to create new data samples that are increasingly similar to the real ones.
- GANs have been used for a wide range of applications, including generating realistic images, synthesizing speech and music, and even creating 3D models.



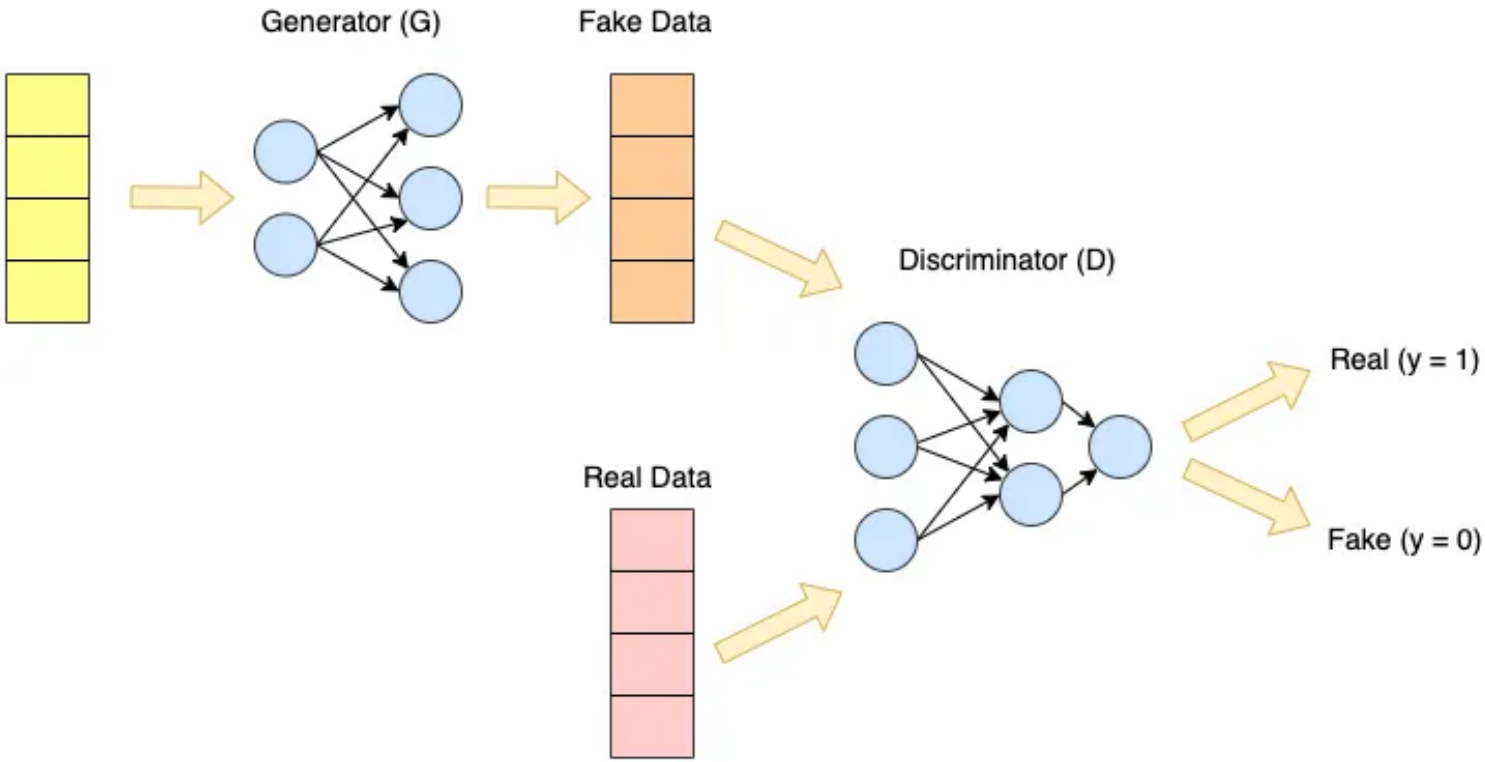
Generative Adversarial Networks

- **Learning Mechanism**
- Both the generator and the discriminator are trained separately during the training process.
- **Training discriminator**: During the training phase of the discriminator, it accepts both real and fake data as input and attempts to differentiate between the two types.



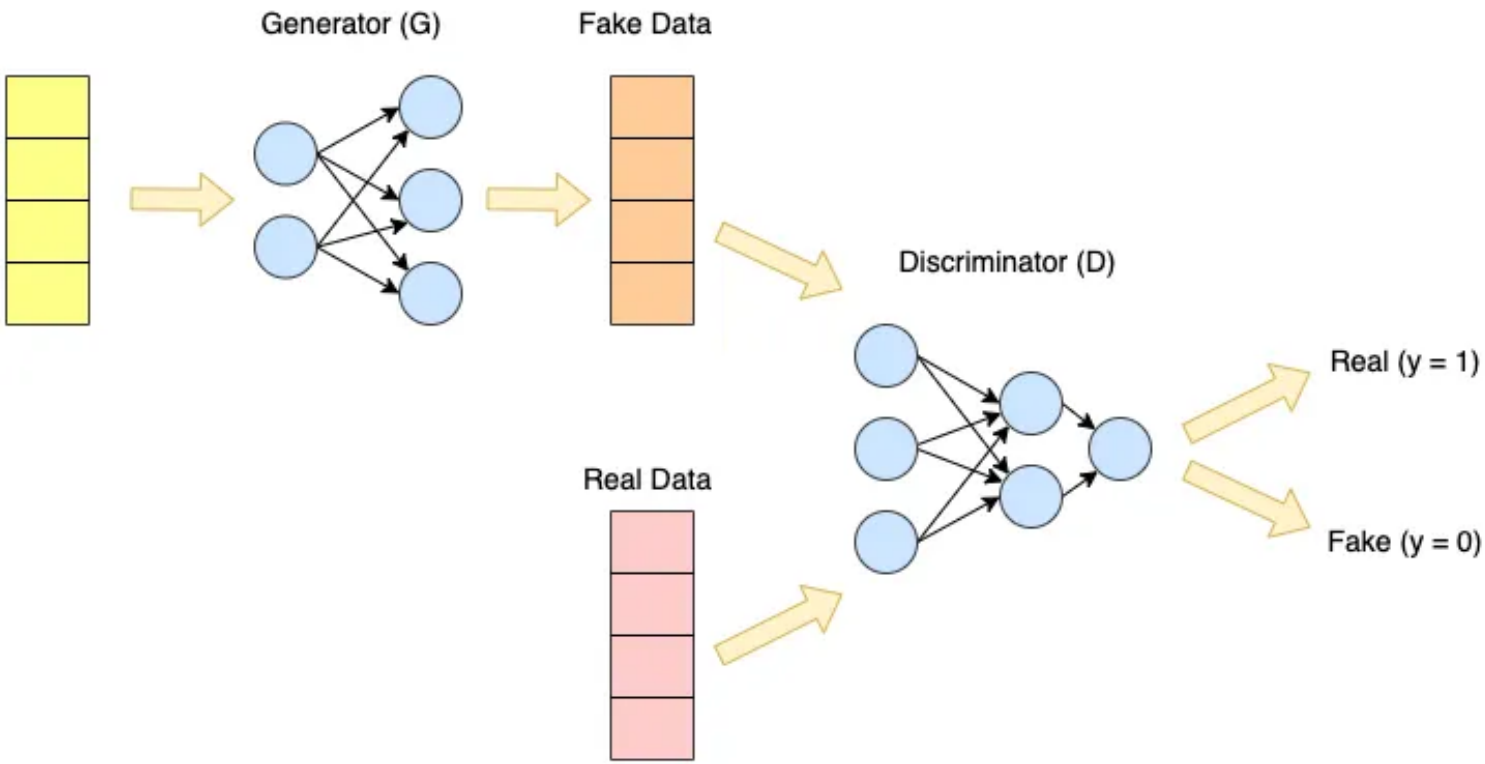
Generative Adversarial Networks

- **Learning Mechanism**
- **Training discriminator**: Training a discriminator is quite similar to training for any other type of binary classification task, in which the network learns to predict zero for fake samples and one for real samples. The generator will be in evaluation mode while the discriminator is being trained. So, only the discriminator's weights are modified during back-propagation based on the loss.



Generative Adversarial Networks

- **Learning Mechanism**
- **Training generator**: Weights in the discriminator are freezed during generator training (that is weights will not get updated). An input random-noise distribution is used by the generator to produce an artificial instance (false data) that is then fed into the discriminator.



Subsection 7

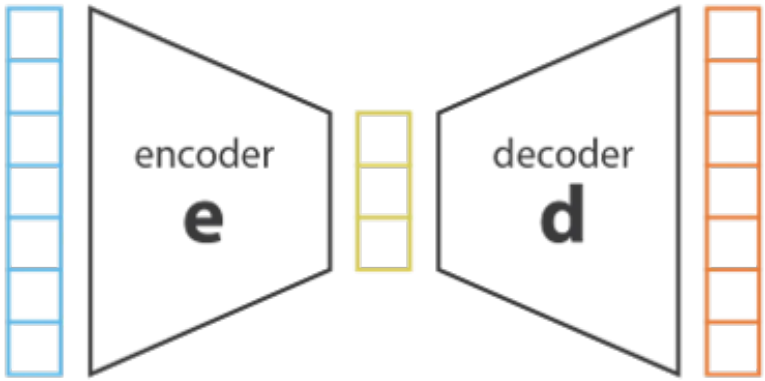
Appendix

Dimensionality Reduction

- In machine learning, **dimensionality reduction is the process of reducing the number of features that describe some data**.
- This reduction is done either by selection (only some existing features are conserved) or by extraction (a reduced number of new features are created based on the old features) and can be useful in many situations that require low dimensional data (data visualisation, data storage, heavy computation...).

Dimensionality Reduction

Let's call **encoder** the process that produce the "new features" representation from the "old features" representation (by selection or by extraction) and **decoder** the reverse process.



x
initial data in space R^n

e(x)
encoded data in latent space R^m (with m<n)

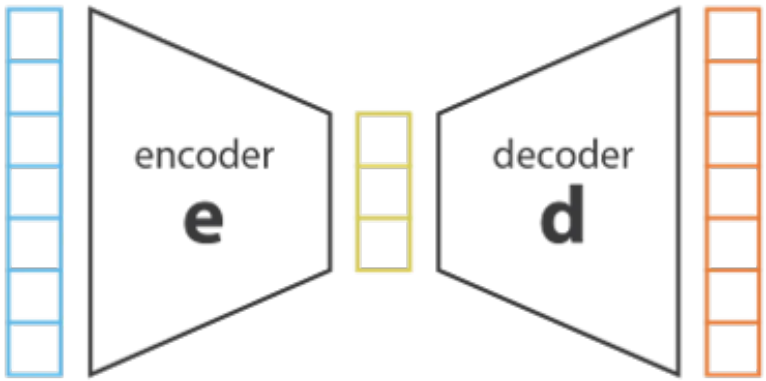
d(e(x))
encoded-decoded data back in the initial space R^n

$x = d(e(x))$ ➡ **lossless encoding**
no information is lost when reducing the number of dimensions

$x \neq d(e(x))$ ➡ **lossy encoding**
some information is lost when reducing the number of dimensions and can't be recovered later

Dimensionality Reduction

Dimensionality reduction can then be interpreted as data compression where the encoder compress the data (from the initial space to the encoded space, also called latent space) whereas the decoder decompress them.



x — initial data in space R^n

$e(x)$ — encoded data in latent space R^m (with $m<n$)

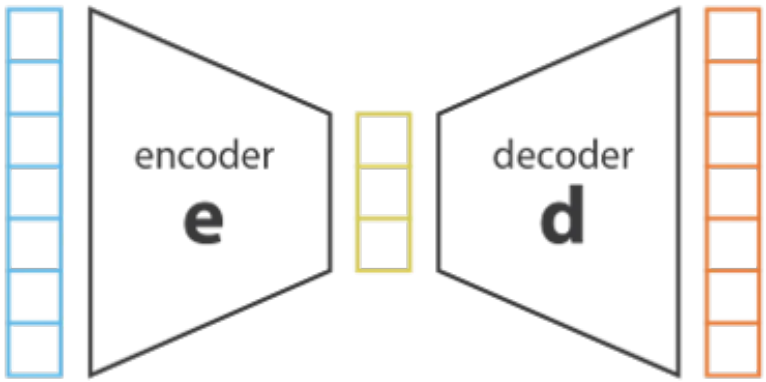
$d(e(x))$ — encoded-decoded data back in the initial space R^n

$x = d(e(x))$ ➡ **lossless encoding** no information is lost when reducing the number of dimensions

$x \neq d(e(x))$ ➡ **lossy encoding** some information is lost when reducing the number of dimensions and can't be recovered later

Dimensionality Reduction

Of course, depending on the initial data distribution, the latent space dimension and the encoder definition, this compression can be lossy, meaning that a part of the information is lost during the encoding process and cannot be recovered when decoding.



x — initial data in space R^n

e(x) — encoded data in latent space R^m (with m<n)

d(e(x)) — encoded-decoded data back in the initial space R^n

$x = d(e(x))$ ➡ **lossless encoding** no information is lost when reducing the number of dimensions

$x \neq d(e(x))$ ➡ **lossy encoding** some information is lost when reducing the number of dimensions and can't be recovered later

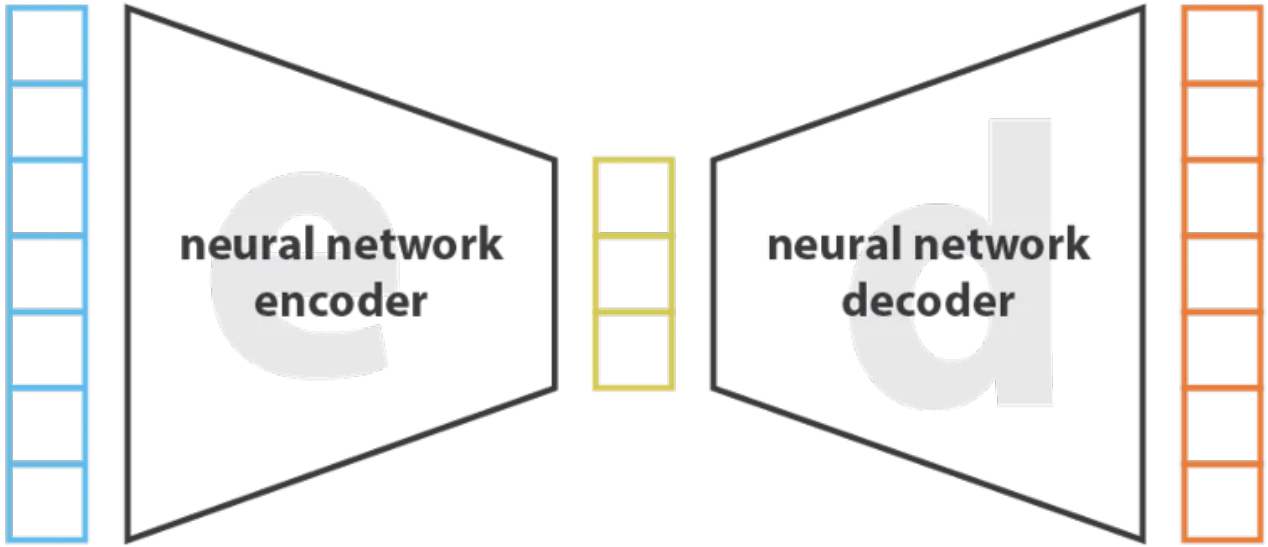
Dimensionality Reduction

- The main purpose of a dimensionality reduction method is to find the best encoder/decoder pair among a given family.
- In other words, for a given set of possible encoders and decoders, we are looking for the pair that **keeps the maximum of information when encoding** and, so, **has the minimum of reconstruction error when decoding**.

Dimensionality Reduction and Autoencoders

- The general idea of autoencoders is pretty simple and consists in **setting an encoder and a decoder as neural networks** and to **learn the best encoding-decoding scheme using an iterative optimisation process**.
- So, at each iteration we feed the autoencoder architecture (the encoder followed by the decoder) with some data, we compare the encoded-decoded output with the initial data and backpropagate the error through the architecture to update the weights of the networks.

Dimensionality Reduction and Autoencoders



$$\text{loss} = ||x - \hat{x}||^2 = ||x - d(z)||^2 = ||x - d(e(x))||^2$$

Dimensionality Reduction and Autoencoders

- Thus, intuitively, the overall autoencoder architecture (encoder+decoder) creates a **bottleneck** for data that ensures only the main structured part of the information can go through and be reconstructed.
- The family **E** of considered encoders is defined by the encoder network architecture, the family **D** of considered decoders is defined by the decoder network architecture and the search of encoder and decoder that minimise the reconstruction error is done by gradient descent over the parameters of these networks.

Variational Autoencoders - The Loss Function

- A variational auto-loss encoder's loss function consists of two terms: one for the reconstruction loss and the other for the K-L divergence;
- Mathematically, the objective function of an AE is given by:

$$L(x, x') = ||x - x'||^2$$

- Where x is the input data, x' is the reconstruction of the input data and $||.||$ is the **L2** norm.

Variational Autoencoders - The Loss Function

- **The Loss Function**
- The objective function of a VAE is given by:

$$L(x, x^{p}rime, z) = ||x - x'||^2 + D_{KL}(q(z|x)||p(z))$$

- Where x is the input data, x' is the reconstruction of the input data, z is the latent variable, $q(z|x)$ is the encoder network, and $p(z)$ is prior on the latent space, typically assumed to be a unit Gaussian distribution. $D_{KL}(.||.)$ is the **Kullback-Leibler divergence**.

Subsection 8

Credits and References

Credits and References

- Renu Khandelwal, Convolutional Neural Network(CNN) Simplified
- https://medium.datadriveninvestor.com/convolutional-neural-network-cnn-simplified-ecafd4ee52c5

Credits and References

- Nafiu, Stock market prediction using LSTM; will the price go up tomorrow. Practical guide (https://medium.com/@nafiu.dev/stock-market-prediction-using-lstm-will-the-price-go-up-tomorrow-practical-guide-d1df2d54a517)

Exercises

https://sh-tsang.medium.com/tutorial-a-good-toy-dataset-for-lstm-model-89e99063610c